# Department of Computer Science


# Technical Report

Undergraduate Dissertation: Grammar Based Modeling and Generation of Tabla Compositions

Joshan Mahmud

**Technical Report 2006-09** **June 2006**

**ISSN 1740-9497**

**Contact Address:**
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

**ISSN 1740-9497**

# GRAMMAR BASED MODELING AND GENERATION OF TABLA COMPOSITIONS

Submitted by Joshan Mahmud
for the degree of
BSc (Hons) Computer Science
2006

**GRAMMAR BASED MODELING AND GENERATION OF TABLA COMPOSITIONS**

Submitted by Joshan Mahmud

COPYRIGHT



**Declaration**

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.....(Joshan Mahmud)

# Abstract

Due to the tradition of oral notation for music and rhythm in Indian Classical Music, implies there is a relation to linguistics. Formal methods which model natural language could be used to model the language of rhythm in Indian classical music, more specifically Tabla compositions. By doing so, a grammar can be developed which can encompass the generation of the entire language and therefore develop the ability to compose Tabla compositions. This highlights the rigorous and strictness in Indian classical music which takes a student many years of experience to gain an appreciation of. If a grammar can be developed from the outcome of this project it will only be able to generate a subset of the entire language since a full understanding of Tabla compositions will not be achieved in the time given. However if a grammar can be developed, then it shows the ability that Tabla compositions can be generated like a natural language and can provide the basic grammar which can be extended to create a fuller language.

# Acknowledgments

I would like to thank my project supervisor Professor John Ffitch for his help and guidance on this project. I would also like to thank my family for their love and support. I would like to make a special thanks to my brother-in-law, Matthew Strevens for proof-reading.

I would also like to thank David Courtney and Jim Kippen for offering their help in my understanding of Tabla compositions and evaluating outputs.

I would also like to thank my house mates whose help and support have aided the completion of this project.

*"Imagination is more important than knowledge."*

**Albert Einstein**

# Contents

# Chapter 1

# Introduction

*"He who makes a mistake is still our friend;*
*He who lengthens or shortens a melody is still our friend;*
*But he who violates the rhythm unawares can never be our friend."*

**-Arabian Proverb**

Indian classical music is one of the most oldest and unbroken traditions in the world. It's origin stem from the Vedas, the ancient script of the Hindus. Presently, Indian classical music is based upon two main elements: Rag and Tal. Rag is the melody of the music while the Tal is concerned with the rhythm. This project aims to deal with the idea of rhythm in Indian classical music, specifically the control of rhythmic patterns using an instrument called the Tabla.

Tal literally means 'clap'. It implies the control of rhythmic patterns in Indian classical music by the use of hand claps. Today the Tabla is used in place of the clap in a performance but the origin of the word remains.

The Tabla is a pair of drums which consists of a small right-hand drum called the Dayan and a larger drum called the Bayan used by the left hand. The instrument is used not to just control the rhythmic patterns but can also create large, complex compositions based upon rhythmic structures and strict formalisms.

One of the most unique attributes of Indian classical music is the oral transmission of music between musicians and listeners due to the lack of formal written notation unlike Western music. As a result Tabla compositions bare a strong relationship to linguistics and natural language.

It is my intention to explore this idea by trying to construct a program which uses formal grammars taken from the study of linguistics as a way in which to model Tabla compositions. This fundamentally implies that Tabla compositions work as a language. David Courtney, an expert Tabla player has made an analogy about learning the theory of Tabla which supports the connection between Tabla compositions and linguistics in [9]:

> *"Just as knowledge of grammar separates a literate speaker from the masses, likewise an understanding of the theory of Indian rhythm and compositional form separates a master from one who is merely functional."*
>
> - David Courtney in Advanced Theory of Tabla

As a musician (mainly in Western music) I have a keen interest in exploring other traditional musical forms and to combine this with being a computer scientist, look forward to investigating this relatively new and exciting area.

The nature of the project implies that a good understanding of Indian classical rhythm will have to be gained before the program can be constructed. The extent to have much can be learnt within the time space for this project will be limited as one who learns who to play the Tabla requires years of experience and dedication. Therefore the system that will be constructed for this project will not be a complete implementation, but serve as a proof of concept that due to the connection to linguistics, Tabla compositions can be formalised by the use of a grammar.

There have been attempts to use formal grammars as a way in which to model musical objects and in particular Tabla compositions, [21]. However I believe that previous attempts have not been able to produce a sufficient solution due to the fact that the theory of Indian classical rhythm had not been formalised appropriately.

The program that will be constructed should allow a generation of several Tabla compositions using the phonetic spelling of strikes on the Tabla known as 'bols'. This should be displayed in the very least a text format and possibly as audio playback. It should be noted that previous research has shown the sound produced on the Tabla for a single bol is context sensitive and therefore may require further investigation. However this is beyond the scope of this project and will not be considered a priority.

The project will finalise by attempting to conclude whether a grammar can be constructed for the generation of Tabla compositions, and if so what it is. In addition, the project should discover what the significance of the grammatical modeling of Tabla compositions tell us about computer generated music and the importance to Indian classical music as a musical tradition.

# Chapter 2

# Literature Review

## 2.1 Indian Classical Rhythm and the Tabla

According to [23], the Tabla in Indian classical music is used as an accompaniment and as a solo instrument. When students used to learn how to play the Tabla there was an extreme focus on its use as a solo instrument and to be able to improvise. Over time it has evolved to become a rhythmic accompaniment to other instruments such as the Sitar, Sarod or vocals which are more melodic instruments.

Despite there being a stronger emphasis on accompaniment nowadays, the Tabla is still widely used as a solo instrument in performance. In order for the Tabla player to gain an appreciation for both the accompaniment and solo playing (compositional and improvisational), they must have a good understanding of the formal structures and compositional techniques that underlie all Indian classical music.

As described by [23], there are set phrases which are used as templates to which both the melodic instrument and the Tabla adhere to when playing together. These compositions are named and are identified by particular criteria which will be discussed in detail later. An example of such a composition would be Alap. It is mainly used for melodic instruments to open a Raag in a slow and non rhythmic way. Another example would be the Vilambit. This section describes the introduction of the Tabla into the piece alongside the melodic instrument. The same idea applies for solo Tabla. There are fixed compositional structures for fixed compositions and improvisations.

As an analogy, one could relate this type of structure to western music, specifically Jazz. There are fixed foundations that underlie Jazz compositions - melodically, harmonically and rhythmically. When played, these structures are used to guide Jazz pieces for the players to improvise over. [1] I wish to discuss the fixed compositional structures which can be played and structures which solo improvisations are based upon.

In order to gain a good understanding of these compositions, we must have a better understanding of Indian classical rhythm and how it is applied in performance. This is an extensive subject and its full discussion cannot be covered here, but I wish to highlight the basic ideas which will contribute to this project.

Books for Tabla tuition make different decisions in the way in which theoretical ideas are taught. This depends on the teacher/ author of the book or the style of playing that is being taught as this is varies quite considerably. Variation of style is mainly due to the geographical origin of the style (Gharanas), [17]. However, from the few that I have read or seen, they all talk about the same fundamental ideas which I will introduce in this paper. [10], [17] are examples of teaching Tabla and are references for the following section.

The first aspect we must look at is the idea of bols. Both [21] and [8] describe bols as mnemonics or onomatopoeic-mnemonics. Bols are a set of terminology which describes the physical striking of the Tabla. Examples of bols are: 'Dha, Ti, Ge, Ta/Na, Ra/Re, Dhi, Ke' whereby each bol maps to a particular way in which to play the Tabla. However, there is no real one-to-one mapping between the bol and a strike as such (no semantic meaning) due to context sensitivity, but I shall elaborate on this point later. When spoken each bol exhibits some oral connotation to the sound produced from the Tabla. Details of how to play bols are found in, [17]. Knowledge of bols are transferred linguistically and are not notated as there is no tradition of written notation in Indian classical music. A good definition of bols is given in [21]:

> *"...An oral notation system using verbal symbols called bols (from the Urdu/Hindi bolna 'to speak') is used for its transmission and, occasionally, performance"*

It would also be a benefit to mention that there are two types of bols - Tabla and Pakhawaj. The Pakahawaj is another percussion instrument which is a predecessor to the Tabla and thus has its own set of bols, many of which overlap with that of the Tabla, [10]. However for the purpose of

[1] One could take this idea further and explore the grammatical equivalences between Jazz and Indian Classical Music since there are many references to their similarities in terms of structure.

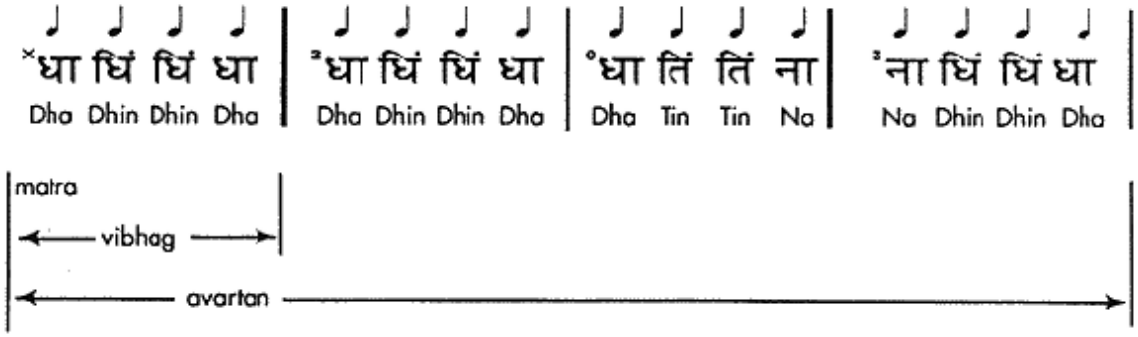



Figure 2.1: example of matra, vihbag and avartan

this paper and for the rest of the project I shall be looking at the Tabla only.

We can now talk about the basic concepts of rhythm and timing in Indian classical music. There are four main ideas which are very similar to western music when looking at Indian classical time theory. These are matra (beat), vibhag (measures/sentences), avartan (cycles) and the Tal system.

Matra is a single beat. It does not specify a unit of time but to a range of tempo based values specified by the player and the piece being performed. Vihbag is a single group of matra much like a bar in western music. The vihbag has a connection to another term which describes the use of stresses on particular beats, other wise known as tali. At the beginning of each vihbag, there is either an emphasis of having a stress on a beat, or having an absence of a stress. Avartan are the cycles that are played, describing the number of vihbags to play. It is the vihbags which denote the Tal system. Different vihbags describe the different time measures that the Tabla can be played in. Figure 2.1 should demonstrate this.

This is the matra, vihbag and avartan in Tal system Teental - a sixteen beat structure spilt into 4 groups whereby there are stresses on the first, second and fourth vihbag. With the use of the crotchets (not to represent a pitch/note but more for its rhythmic value) one could interpret this as a 4/4 time signature in western music. The Tal system is extremely important in Indian classical music as it describes the fundamental rhythmic timing of a piece. There are many Tals that exist, but here are a few more common Tals as show in Figures 2.2 and 2.3.

Each of these patterns which belong to the Tal system are known also as Thekas. For aesthetic purposes, they may be thought of as a groove, or the particular rhythmic style a piece can be

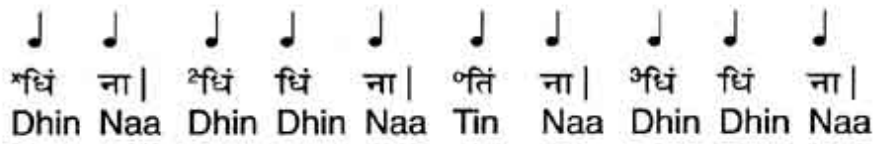



Figure 2.2: Jhaptal

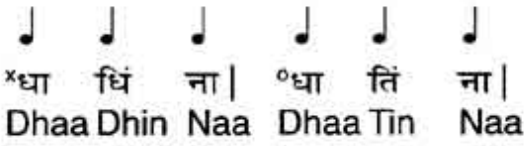



Figure 2.3: Dadratal

played in and are used to accompany other instruments, though now are considered an important part of understanding basic rhythm.

The final basic idea that I wish to highlight is the idea of the Layakari. Once a Tabla player has learnt and understood the fundamental abstract ideas of rhythm and time they must adapt them to performance. When listening to an accomplished Tabla player one may find that they may play at varying speeds. This is because they are exploring the relationship between the basic rhythm of a piece and playing over it. This is known as the layakari. It refers to whether the Tabla player is playing in single time (a beat per matra), double-time (2 beats per matra) etc. It can also extend to playing obscure timings like triplets (3 beats over 2).

Now that we have a firmer understanding of Indian classical rhythm we can now begin to talk about the types of compositional structures that exist for Tabla.

Different books for Tabla tuition describe two approaches that are taken when teaching these compositions. A book like [17] takes a more direct approach and demonstrates the ways in which different compositions can be played without explaining why they can be played in such a way. This seems to be a more traditional way of teaching Tabla as the student gains an appreciation of compositions implicitly through experience. One could think of this idea when learning natural language - the way in which a baby learns to speak English through hearing and gaining understanding over time. This is actually a very deep and important idea which I wish to elaborate slightly more on later when connecting linguistics to music.

If we were to look at a book such as [10], we find that a more detailed analysis has been taken in its explanation of compositions. The purposes may be due to the fact Courtney acknowledges the way in which western readers/musicians understand music and rely on a heavily formulated and structured approach of understanding music.

Courtney classifies Tabla composition into two classes: cadential and cyclical form. Cadential implies that the composition reaches a climax or a point of ending usually on the sam, which is the first matra of a theka.[2] The cyclical form implies that composition moves in a fixed structure, but with no sense of a resolution. There are many compositions under both categories and listing them all would not provide much benefit to this project. Therefore, I wish to highlight only a few explaining their structure and an important point Courtney highlights in particular.

### 2.1.1 Bol, Structure and Function

Courtney makes a further and quite unique analysis of compositions by explaining that they can be defined under 3 separate criteria which are mutually exclusive. The first is the bol - the specific bols that are used in a composition are important. They could be Tabla or Pakhawaj or even other instruments such as a Dholak. The second is structure, the way in which the composition is built, like whether it repeats phrases or has two separate sections or even repetition of avartan. The final criterion is function. This describes the location of the composition in a performance. It could be used as an opening, comprise as the main body of a piece or even transition between two sections. The three criteria do not affect each other and rarely are all three used together to define a composition.

With these three criteria, defining the compositions becomes a simpler task. First we define two compositions from the cadential form: Mukhada and Tihai.

### 2.1.2 Tabla Compositions (Cadential Form)

Mukhada literally means 'face' in Urdu/Bengali, and is defined by structure and function. Thus any bols may be used. It is a short piece and its structure is similar to a Theka, but the last vihbag usually increases in bol density or Layakari creating a sense of tension which resolves on the

[2]The word cadential used by Courtney in Indian classical rhythm is analogous to that of its use in western music when talking about cadential chord progressions, and how they exhibit resolution in tone by ending on the root chord, [2].

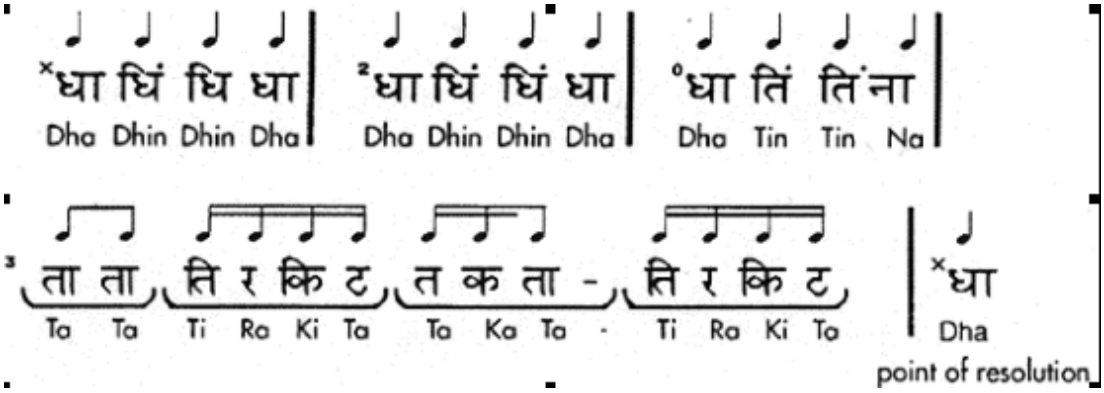



Figure 2.4: example of a Mukhada

Sam. Functionally, it is used at the end of a piece/ section of a piece. Figure 2.4 is a diagram to aid this explanation. It is a Mukhada in Teental.

Tihai is a slightly more complex composition defined only by its structure. It is essentially a repetition of 3 phrases (pala), with the last phrase ending on the sam. Again, it creates a sense of tension and the repetition creates a climax which reaches a resolution on the sam. The phrases are connected in one of two ways. Either each phrase follows another directly (Bedum Tihai), or there is a pause in between the phrases of a fixed number of beats (Dumdar Tihai). Figure 2.5 is a diagram of a Bedum Tihai in Teental.

Courtney provides a mathematical analysis of the Tihai. He explains that the number of beats in a phrase depends on the layakari and the Tal in which a piece is played in. He provides a formula to calculate this (p = phrase, n = number of beats in Tal, l = layakari, d = pause):

$$p = ((n * l) + 1)/3$$

*Bedum Tihai*

or

$$p = (((n * l) + 1)/3) - 2d$$

*Dumdar Tihai*

The fact that the composition can be mathematically formulated indicates the strictness of the composition. Thus from these two compositions it can be seen that there are compositions which are simple and others which are more complex creating a range of aesthetic effects. I would now like to have a look at one more composition from the cyclical form known as the Kaida.

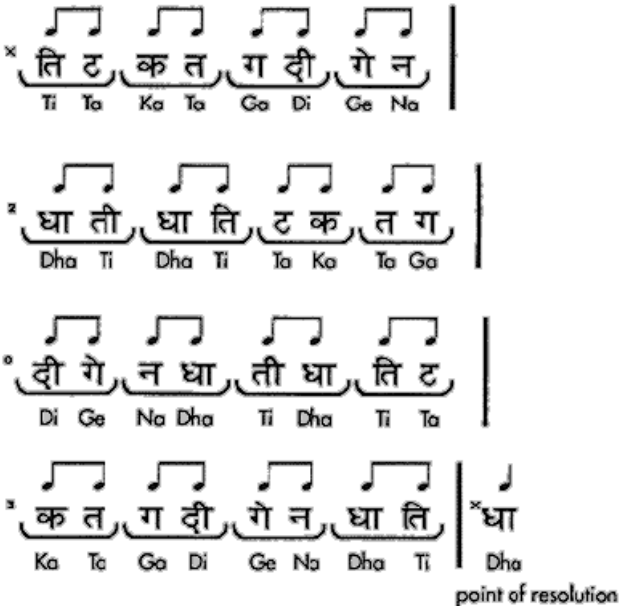


Figure 2.5: example of a Bedum Tihai in Teental

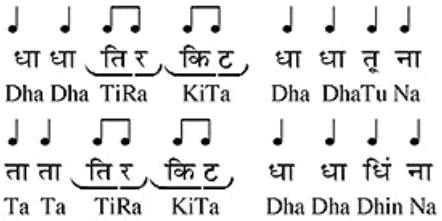


Figure 2.6: Theme of the Kaida

### 2.1.3 Tabla Compositions (Cyclical Form)

The Kaida is a special kind of compositional form which is defined by its structure. It is concerned with the idea of theme and variation whereby a theme is introduced in a piece and what follows are improvisations upon that theme. Although the Tabla player has the freedom to improvise, they are bounded by particular rules attached to the Kaida. This composition is best demonstrated with the use of diagrams as shown in Figures 2.6 and 2.7. One is to describe the theme and the other to describe the improvised variation.

It can be clearly seen that the variation uses substitution, permutation and repetition of the bols from the original theme. However there are rules to which the variation conforms to. The first are the bols. The bols used in the variation can only be the ones used in the theme. Secondly,

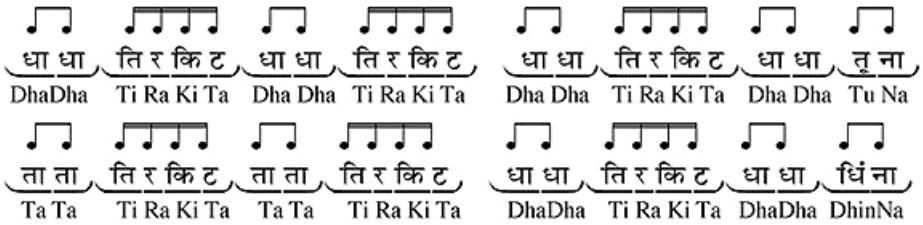


Figure 2.7: Variations of the Kaida

although not shown here, the Kaida must take the form of having an introduction, a body and a Tihai, which uses the bols. Another structural rule is that the Kaida must exhibit a Bhari/Khali arrangement. This means that everything must be played twice. Therefore the variation section can be played twice and then have a resolving Tihai.

Although not mentioned in this paper, [10] mentions the context sensitivity of certain bols. As there is no real semantic meaning attached to bols, similar notation of bols can correspond to different strikes. There are only a few of these bols but here is an example - TiTe. The Ti in this sequence is played slightly differently when used in this context than when used on its own. However when used orally or even notationally as it is used here (despite the lack of formal notation in Indian Classical music) Ti is always written the same.

With these constraints, a number of different variations can take place, many of which can be named. One that Courtney and [17] talk about is the palta. This describes the various permutations of the bols in the theme with a variance of the layakari. We find something very interesting however in Courtney's explanation of the variation section of the Kaida. He ascribes the vihbags of the above theme with different 'codes' based on their rhyming scheme. Thus he associates to vihbags one and three the code A and to vihbags two and four code B. He then shows the valid 'mathematical permutations' of the variation section such as: AAAB-AAAB, or if we break the vihbags down further and assign codes to them we can develop the permutation: CDCAB-CDCAB. By doing this, Courtney has implicitly described a lexicalised computational grammar, by showing valid sentences in a language for the Kaida. With certain constraints and a set of words, he has shown the valid sentences that can be derived. This heavily supports the idea that Tabla compositions could be modelled by a formal grammar.

If we again analyse the idea of bols and the tradition of oral notation in Indian classical music, we realise that there is a strong element which is connected with natural linguistics. Therefore, it could be argued that as there have been attempts model linguistics by grammars, we can attempt

to model Tabla compositions with grammars too. The problem lies in the fact that we are trying and define a set which encompasses all possible valid sentences of the bols. Dr Jim Kippen attempts to explain this is further in detail [19]. He argues, with specific reference to the Kaida, that although permutations are allowed, they are not completely random and there is a set of governing rules which allow certain bols to appear at specific rhythmic locations. [18] talks about a specific style of playing the Tabla, which originated from Lucknow in North India. Kippen talks about the idea of the Khula/Band type bols. Khula and Band mean resonant and non-resonant sounding bols, respectively. Kaidas in various Tal have a particular format whereby the bols that are used at particular locations are either Khula or Band bols. These locations are the Bhari and Khali sections of the Kaida. The Bhari section uses Band bols while the Khali section uses Khula bols. [21] mention the string of voiced and a matching string of bols which contains some of the original bols to be transformed into their corresponding band bol, thus creating a homomorphism between the set of bols mapping onto itself. These bols are: dha/ta, ge/ke, dhi,ti. This concept can be highlight better by comparing Thekas to the themes of Kaidas. If one analyses the areas in which the Khula bols are used in a Theka and the corresponding Band bols, the same pattern should follow in the theme of the Kaida. Here is a structure of the Khula and Band bols of theme of a Kaida in Teental ([18]):

**X Khula Khula Khula Khula**

**2 Khula Khula Khula Band**

**0 Band Band Band Band**

**3 Khula Khula Khula Khula**

[19] mentions that as there are building blocks in the construction of the Kaida, we can construct a dynamic tree model, which could produce valid sentences for a given grammar with string rewriting rules. Kippen goes on to talk about the Bol Processor, a piece of work which I wish to elaborate on later. Now however, I wish to talk about the types of grammars which would be useful in modeling Tabla compositions and existing grammars that attempt do this already and try to argue their ability in producing valid sequences/ patterns.

## 2.2 Computational and Linguistic Grammars

In the paper previously referenced, Kippen mentions that the variations that are produced which are in a Kaida are up to the Tabla player. Both [10] and [17] mention that a student would spend many years listening to their teacher and practicing basic variations, building that variance of improvisation over time. We can again draw an analogy with natural language and look at this idea from the perspective of psycholinguistics. [27] talks about the way in which a baby may learn natural language by listening to spoken word of others and gradually gaining an understanding of valid sentences. They may begin to exhibit understanding by either responding to natural language or by attempting to communicate - initially with crying or sounds from the mouth. This may develop into as Scovel describes, 'coo-ing', utterances which may not be sound or be perceived as a valid word, but may have semantic meaning. Finally they may begin to speak actual words or phrases which are valid. A Tabla student may develop the 'natural language' of improvisation and composition in this manner making their cognitive process more natural to their own thinking. It seems that one of the first attempts to try and model Tabla bols is by Dr Jim Kippen and Dr Benard Bel in a way which may somewhat conform to this idea of gaining experience by building an expert system. [20]. I shall talk about the actual work produced later when reviewing existing work related to this project. For now I wish to talk only about the grammar that was produced.

### 2.2.1 Pattern Languages

Kippen and Bel transcribed a theme of a Kaida and variations played by a Tabla master, Ustad Afaq Husain Khan. The aim was to develop a machine learning system, based upon a finite state automata, which would take an input stream and after analysis would have a set of rules defining states and the bols between transitions of states. The machine would gain experience by generating a sequence based on a start symbol, and structure the automata graph so that it would terminate. An expert would check whether the sequences were valid or not. If not, amendments would be made to the system to increase the rule set gain further experience. Due to the limitations of the finite state automata, and the problems of having context sensitivity of particular bols it is not a stable solution. It also relies solely on the input data. The machine cannot identify more variations than supplied by the input and cannot make 'artistic' decisions.

[3] develops this idea further for pattern matching languages insisting on the fact that for a Kaida, there is a set of all variations that are valid and a set which are not. It shows that the set of valid variations is finite as for a given Tal, as the maximum bol density is when the layakari is 6,

though this could be greater if a Tabla player was able to play extremely fast. Pattern languages are a natural way to think of producing Tabla bols as the sequences are patterns with constraints. However, the constraints pose the problem in producing valid sequences after observing a string of valid sequences. Pattern languages are also argued to be a good way in which to model a stream of bols due to the context sensitivity of groups of bols. Data which accumulates small groups of bols segmented in a meaningful way, the issue of this context sensitivity becomes reduced.

We must realise that when a student learns the Tabla, they only hear valid sequences (referred to as the positive context by [3]. Pattern languages could produce anything from the positive context or the negative context. Thus other methods are explored such as the restricted pattern language grammar. However, the restricted pattern language is only a 'descriptive generalisation' which is not necessarily a model. It does not describe all valid sequences, and only the structure of the sequence of bols it analyses.

If we took this idea further and attempted to observe the pattern or the hierarchy in the sequence of valid bols we could use something like the SEQUITUR algorithm, [26]. This algorithm identifies a hierarchical structure that may occur in natural sequences, and has been demonstrated in linguistics and music. The heart of the algorithm lies in the fact it attempts to identify the linear repetitions of words in a sequence, create a unique production rule and arrange the rules into a hierarchy. This hierarchy is like the restricted pattern language as it is only a structure description of the sequence it was given and not a model of the underlying structure. Although the algorithm is efficient and not necessarily a correct interpretation of the underlying structure, it may introduce interesting structures that occur in Tabla improvisation by a single player. As previously discussed, styles of playing Tabla differ depending on the geographical origin yielding different ways of producing improvisations. Though again this algorithm may transcribe an input sequence of Tabla bols into some natural hierarchical structure, it may not be structurally equivalent to that of the hierarchical structure in Tabla compositions and Kaida improvisations. Instead, it may yield more subtler effects in different styles of playing which could used as a basis of comparing different styles.

As the pattern language could still have the capacity to produce sequences in the negative context as it has been proven in [3] that a pattern language is not closed under the set operations of union, complement and intersection. Thus the set of sequences produced by the pattern language is the union sequences of the positive and negative context sets. Kippen and Bel developed their ideas on pattern languages and phrase structure to create their own grammar to model improvisations

in Kaida known as the Bol Processor grammar, [21].

### 2.2.2 Bol Processor Grammars

This grammar is presented as a possible solution to the idea of modeling Tabla improvisation in Kaidas grammatically. Due to the natural simplicity of pattern grammars and due to the author's lack of knowledge of Tabla improvisation, modeling Tabla Kaidas as pattern languages seemed to be a pragmatic idea to form a basis for implementing this grammar. Although I will not go into detail about the way in which this grammar works, I wish to highlight the parts which I consider significant.

One of the main things that the Bol Processor grammar identifies are the types of permutations of bols and also identifies other constraints, such as the Khula/Band homomorphism. A compact and efficient use of notation is used to identify when strings of bols are either equivalent or copies of other bols. If they are copies, other permutations or transformations can be done on them allowing a formalised way in which the variations take place.

It also uses the idea of not just having single bols as terminal symbols in production rules, but groups of bols. This is because there is a stress of particular bols which after analysis, may denote the style of playing or a particular sequence which cannot be subdivided. This also implies that this particular group of bols is used more often together than separately. Groups of bols also imply the layakari/ bol density that is used and thus the grammar does not have control over varied laykari.

Another aspect of the grammar is that it identifies the hierarchical structure of the bols and thus the grammar is structured in such a way that it comprises of multiple sub grammars. They are executed in a linear order, whereby terminals in one grammar can be non-terminals in the subsequent grammar. This shows that there is a phrase structure of some description that lies behind the derivation which needs to be established before strings of bols can be outputted which is done on the last grammar. The grammar does take into consideration the context sensitive derivation of particular bols, and highlights such constraints:

1. there should be no more than two consecutive dha;
2. there should be no more than two consecutive '-'; (blanks)
3. ti should not appear before or after ti or trkt.

These constraints are specific to the style of playing that Kippen and Bel based this analysis on. This instantly implies that the grammar derived is specific only to this style or even to the single Tabla player who generated the original variation for analysis. Thus, improvisations modelled by this type of grammar are bound to a particular context and constraints which may be difficult to extend or alter making it inflexible. However, it does use other tools to create a sense of a phrase structure such as bracketing or structural markers, but these too are context sensitive and are bound to a specific meaning such as a bracket '(' followed by a '+' implies a group of four lines are to follow. This would not be valid if derivations were in Jhaptal as opposed to Teental.

Although the grammar described may provide essential tools to model an improvisation, the paper acknowledges the fact that it may not identify irregularities and that modeling all Kaidas may not be possible in a formal representation such as this one. But it may also contribute to the way in which knowledge is passed to the system which is at fault, rather than the formalising method. Although this may be true, I believe that it can be argued that the model described is too focussed on the specific composition as shown by my reasons previously. Thus due to its inflexibility and inability to handle irregularities a more dynamic approach is needed. Another aspect of the Bol Processor Grammar is its ability to be used to recognise sequences. As there is a linear format to the execution of the sub-grammars, it allows parsing to work by reversing all of the production rules, thus all production rules are bi-directional. We can also highlight the fact that other compositional types and varying improvisational techniques require different grammars to that of shown in [21]. In order to embrace the other compositional types and attempt to model them as well as the Kaida, we may need to look at the way in which compositions are made and identify computational grammars which can model this correctly.

One of the first things we can identify is that there is a phrase structure to Tabla compositions. Each of the compositions can be thought of as a phrase by which bols are attached to it depending on where it fits in to the whole structure. Therefore we start by looking at a Phrase Structure Grammar.

### 2.2.3 Phrase Structure Grammars

The phrase structure grammar originated from a work by Noam Chomsky, [7] entitled Syntactic Structures. This work provides a deep analysis of natural language, focusing discussions by analysing the formal structures within the English language. The idea of the phrase structure grammar was born out of the desire to control and formalise the way in which derivations occurred in natural languages strictly in the positive context. This has given rise to the Chomsky

hierarchy of grammars. This idea moves away from having pattern languages and to have the ability to analyse the formal structures beneath a language, modeling them as a set of production rules based on the language semantics. We have seen that the Bol Processor grammar combines a pattern language with the concepts of a phrase structure grammar, as is shown by its use of multiple sub-grammars. However as mentioned by [21], if the produced strings are not quite correct or do not create what is required, rather than the model being incorrect, the transmission of information may be incorrect. However, the transmission of analysis depends on the modeling type which effects how we interpret information for the model to understand. Thus, if we have a better or alternative understanding of the phrase structure then inferring it may yield different results closer to the concept we are trying to model.

The question we must ask therefore, if we have a particular understanding of the grammar and structure of the language we are attempting to model, then is the phrase structure grammar powerful enough to do so.

In a revision of [7] is[22], Lasnik talks about the uses of the phrase structure grammar. Although this is a wide discussion, there are a few vital points that need to be made. Firstly is the idea of structure and infinity. The phrase structure grammar helps us to control the structure of a language and the infinite derivations that can occur. Natural language has a grammar which allows infinite derivations. We can argue that we would want a grammar that composes Tabla bols infinitely, but we can make a decision not to have this as we would only want (for now), a finite piece composed. It could be argued that this is an artistic decision whether to have infinitely long compositions, but from [20], we know the set of valid sequences is finite thus we would want to reach a terminal symbol.

Secondly we can look at how useful the grammar is when lexicalised. This means looking at the way in which the grammar works with a word/verbal-based morphology. As previously mentioned in [20], the vocabulary used in Tabla compositions, i.e. bols, do not have any real semantic meaning. Bols do not have semantics attached to them individually and are more context bound. This raises the issue on its morphology. This implies that swapping the ordering of bols in a particular phrase may not alter it's 'meaning' as such but is interpreted in a musically aesthetic way. However, we still have the issues of constraints of Tal, layakari and context sensitivity in compositions. We cannot make the exact parallels in Tabla patterns and linguistics, though there are some connections that can be identified. In essence, can we make generalisations about the syntax of the Tabla bols in compositions?

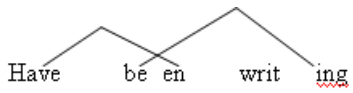



Figure 2.8: Example of cross dependencies within the English language

Laskin uses an example originally illustrated by Chomsky in the English verbal morphology, whereby making generalisations of particular sentences are not powerful enough in a phrase structure grammar. This is concerned with the idea of cross-serial dependencies. Lasnik explains that phrase structure grammars are very good at nested dependencies. The example given in [22] as shown in Figure 2.8.

The example shown above is a quick example of the way in which the morphology of the words is dependent on other words. A similar issue can occur in Tabla compositions. If we look back at the example of the Kaida, we find that we have a theme and numerous variations upon it. Here we can see that not only are there cross-serial dependencies between the bols of the variations but that also the constraint of the having the Khula/Band bols in the correct locations.

In order to resolve this issue we must find a more powerful grammar. Available to linguistics are two grammars which I think are appropriate to mention due to their unique attributes. The first is the transformational grammar.

### 2.2.4 Transformational Grammars

This grammar is still part of [7] and is a suggested solution to the problem stated above. This is an extensive subject and a full discussion of it here is not possible. In essence, the transformational grammar has 2 main properties, the structural index and the structural change. It attempts to identify an abstract structure in the syntax tree/ representation (syntactic/structural index). This tree representation is constructed by a phrase structure grammar. Depending on the type of transformation that is required, it specifies the change that needs to be done to it in order to make it a valid sentence (syntactic/structual changes).

The idea of relating transformational grammars to music was first exposed by Leonard Bernstein in his televised Harvard lectures known as “The Unanswered Question”. Although I have not been able watch these lectures, there is a good overview in a discussion by David Bernstein and Tom Strychacz, [5]. He explains that there is a surface and deep structure in music and that

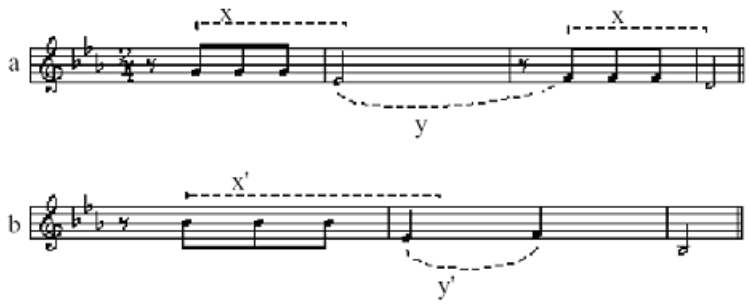



Figure 2.9: Deep structure connection between melodies of Beethoven's Fifth Symphony

there is a universal musical language.

This is open to interpretation, but one of the ideas which he explains I feel is very important to this discussion is the connection between linguistics and music. He uses the idea of a metaphor used in natural language to describe aesthetics in language, specifically in poetry. For example, if we had the sentence: "Juliet is the sun", it contains semantics which are different to that of being used in every day natural language. In music, a metaphor could describe a connection between two melodies (intrinsic metaphor). An example that is used is the opening of Beethoven's 5th symphony as shown in Figure 2.9.

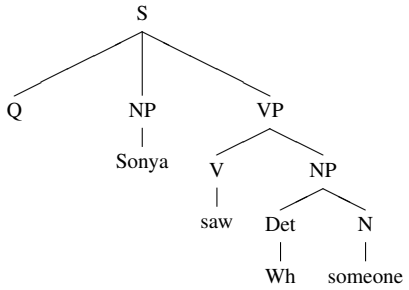
As it can be seen, melody 'a' has a repetition of melody with a change of pitch (y). In 'b' which occurs later in the piece, the melody has altered, but bares relation to the initial melody. This connection is argued by Bernstein that there is a deeper abstract/ musical connection, though the surface structures appear to be different. Although this is an 'aesthetic' deep structure, we can still relate this to a more concrete deep structure.

To relate this to Tabla compositions, we can find that repetition or use of bols in 2 different Tihais for example can be structurally equivalent. The use of transformational grammars may allow us to identify structures and perform transformations on them which we configure as valid to yield a set of bols that can be played. Since we have control over the transformations, we can specify the cross - serial dependencies and thus control the bols created and rhythmic locations. This implies that we could have control over the use of Khula/ Band bols at correct points in a Kaida.

In a different interpretation one can look at the similarities in the language of Tabla compositions and natural linguistics. The transformational grammar has been developed due to the problem

of cross dependencies as explained in the discussion of the phrase structure grammar. It allows the alteration of the phrase structure in a cross serial level and allows the manipulation of the structure of sentences beyond the scope of the phrase structure grammar. Therefore, one can construct many interpretations of a single idea modeled by a phrase structure depending on the transformational rule. If we observe literature on Indian classical music, there is the repeating notion that it uses no counterpoint and is based on a cyclical structure producing a greater hypnotic mood. One could think of this in terms of linguistics as a basic phrase structure with many variations whereby the variations are some interpretation of the semantic or aesthetic definition of a composition. Transformational rules could define these variations upon some phrase structure of a Tabla composition producing a powerful way in which the language may be controlled.

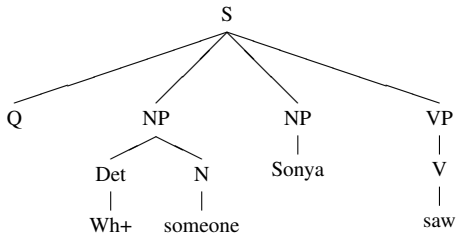
Although this is a powerful tool, we may find that this could be unmanageable. If we wish to perform transformations on structures that already have had transformations performed on them then calculating the structural analysis could become exponentially large. For example, if we wish to have a sentence which has: '**Dha Dha Ti Te**' and wish to perform a transformation: '**DhaTi DhaTi Ti Te**', and then wish to develop this as another variation later, we would have to realise the current structural analysis, i.e. the rhythmic timing of each bol. This problem does not occur with natural language as the phrase structure can be labeled with the semantics of the words of the language and therefore the transformations are based around the manipulation of the phrase structure based upon these semantics. For example a basic phrase structure for the sentence: "Who did Sonya see" could be:




A transformation in table 2.1 shows a 'Wh-Q Movement' transformation to create the sentence "Sonya saw someone" [14], shown as follows:

Table 2.1: Wh-Q Movemement Transformation Rule

| Wh-Q Movement | Q | X | Wh+N | X |
|---|---|---|---|---|
| Structural Index | 1 | 2 | 3 | 4 |
| Structural Change | 1+3 | 2 | NULL | 4 |




A further transformation would be needed to insert the 'Do' word, but the example highlights the transformation based upon the semantics of words in the phrase structure. The 'words' in the language we are attempting to model do not have these kind of semantics, despite the cross serial dependencies. Therefore there I wish to look at one final grammar which may help this situation, the tree-adjoining grammar.

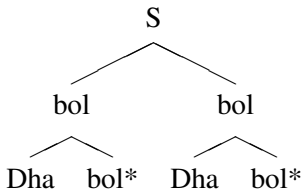
### 2.2.5 Tree Adjoining Grammars

The use of tree adjoining grammars in modeling linguistics has been used as an alternative to transformational grammars. In essence, its main difference is that rather than generating a string sequence it generates a parse tree, [1].

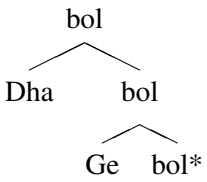
It has been argued that a lexicalised tree-adjoining grammar has the capacity of representing a context free grammar and thus has the ability to model a natural language, [16]. If a tree-adjoining grammar is lexicalized, it implies that the tree has lexicons or "anchors" attached to the leaf or foot nodes of the tree. The grammar uses two types of transformations which have been adopted from transformational grammars to generate the tree: substitution and adjoining. Although the discussions about the power of the tree adjoining grammar attempting to model natural language, e.g. the XTAG system, [13], I do not wish to discuss this. This is because I do not believe the tree adjoining grammar to be an efficient way to model a language. It is highly centered around the lexical anchors in a language which is based on the semantics of words. I do not believe that this would be an appropriate choice for Tabla compositions though there can be attempts to do so.

Instead I am focusing on a property of the tree-adjoining grammar whereby all elements are treated as trees.

If we were to model each lexical item as a tree we can implicitly control the context in which it can be used and also specify the domain of locality for it. Therefore if we were to have a string of bols: 'Dha Dha' we could model each as a tree:




If we use the adjoin transformation we would be able to increase the number of bols attached to each existing bol. This would instantly describe the number of bols played per matra (from observing Layer1). Therefore we could adjoin another bol 'Ge' on to the right hand tree:




We would not be bounded to on bol either. Instead of just 'Ge', we could have 'DhaGe', thus putting more emphasis on the pattern 'DhaGe' occurring more often together.

After discussing these grammars we can say that the phrase structure of Tabla compositions is essential to the compositions derivation. However due to the cross dependency issues, we must use something more powerful such as the transformational grammar. However, as we are not looking at semantics of lexicons and verbal morphology, we are more concerned with having control of the rhythmic timing and generation of individual bols as well as groups of them and would require of another structure to handle this. The tree adjoining grammar could help with this and allow us to model the use of adding bols in a semantically rich way specific to a tree structure after the construction of the phrase structure. We can see that grammatical modeling of Tabla compositions could use ideas from existing grammars such as the Bol Processor grammar and be an intersection between other formal grammars in order to correctly model its internal and external structures.

## 2.3 Grammar Based Music Composition

This is a large area of work which has been explored by computer scientists and musicians alike. Similarly there are many areas which branches from computer music composition. There are expert systems which attempt to simulate players, improvisation or the computer is used to compose complex pieces. This work has been mainly western music related, and deals with many issues that we are not concerned with in this project such as pitch, melody and harmony.

Since the music we are trying to generate is monophonic many of these issues we do not have to consider, and thus the creative process may be affected in the style and type of music we are trying to compose.

So far we have been looking at Tabla compositions at a low level in particular its construction. Whether this represents the cognitive process of a musician is debatable. I have made no assumptions that this is the natural process of creativity and although the way in which we may represent a model grammatically may be true, this may not yield 'creative' or what is considered to sound 'good' in an inference process.

This comes to the idea of whether the idea of creativity is computable. Creativity could be argued to be the product of our lives and experience, under the study of anthropology. However in order to attain "good" compositions, we must first be able to create valid compositions.

A grammar allows us to make derivations which are valid, but we can constrain this to making informed decisions and therefore generate not just valid derivations but ones which give an aesthetic influenced one, [25]. One of the simplest ways to do this is to use a stochastic process. This is when given a set of elements, a set of probabilities are mapped to these elements creating a probability distribution in the selection of them. A set of probabilities are mapped directly to a set of production rules and the selection of production rules are reliant on their associated probability. This implies that the combined probabilities must equate to 1. The calculation of the probabilities can be done by using a Markov model as explained by McCormack and [29]. McCormack makes other interesting points such as the use of parametric extensions whereby real values correspond to terminals or non-terminals in the grammar. These values are passed between production rules and arithmetic operations can be done on those values in corresponding to different symbols.

Other more advanced options that were considered such as use of grammatical evolution in

choosing the 'best' solution, [12]. I would think that this method could be implemented at a stage after this initial inference section has been implemented as once valid sequences are clarified, optimising this further would be the next step and could yield a better derivation of compositions.

## 2.4 Current Work

The work most related to this project is one that has been previously mentioned but in brief: the Bol Processor, [21]. The work was born out of the idea of attempting to model the improvisational techniques that occurred in Kaidas. The application comprised of an editor designed to represent bols. It had an inference engine which would use stochastic processes to generate bols using the grammar defined, and and membership method which would use the grammar in reverse to parse compositions to check whether they were correct given the grammar.

I have already commented on the grammar that was implemented, thus the only other aspect that I wish to mention is that although the research found that a grammar was produced which mimicked variations well, there was no more exposition to other composition types.

The application was an expert system thus the grammar was hard-coded into the application. Therefore changes were being made to it in order to improve the accuracy and the requirement of a more abstract system was needed whereby the grammar was modular and thus could be altered without altering the actual application. This gave rise to the BP2 application developed by Professor Benard Bel, [4].

In terms of linguistics, there has been numerous works on using grammars to model natural languages. As shown in [7], the author attempts to model English and shows a set of transformations for the language. The XTAG project which again was previously mentioned is an existing, and still developing grammar attempts to model the English language. However due to the fact that the grammar is heavily lexically based (all elementary trees - union of terminal and non-terminal trees, have a lexical anchor), the grammar is extremely long, and management of it is hard. Solutions have been proposed by use of a hierarchy which is linguistically driven, [24].

As a usable application for practical purposes, there is an application known as the TaalWizard, [15]. This application is extremely user-driven and gives a rich interface whereby bols are played back to the user and a graphical representation much like a sequencer is displayed.

## 2.5 Conclusion

I wish to begin an implementation of a grammar taking into consideration the issues I have raised in this literature review and begin to formulate an idea of the grammar by gaining a deeper understanding of the Tabla compositions. I am confident that the environment I will be working in will be C or C++ due to the power of the languages and the necessity to use a language which is familiar to the author.

I shall also be using the ideas of TaalWizard as a basic idea of a front end whereby the bols are represented on screen and are played back to the user by use of samples. This will be done by looking at a basic graphic package for C/C++ known as Qt. This is a simple package to create rich and basic interfaces with a short learning curve. As the primary object of this project is to develop a valid grammar, less time will be planned for improvements on the graphical interface.

It has been shown that a good understanding of Tabla compositions is required in order to produce a valid grammar to model compositional types. The requirement to use a Tabla expert to validate compositions will be useful and can be used as a source of asking for additional information. Professor John Ffitch has had some experience of playing the Tabla but has referred me to seek further help from known Tabla players such as David Courtney.

I have also discussed that using a grammar type that already exists in a strict form may be too limited and that a solution will need to be produced which uses an intersection of existing grammars.

I wish to make the final application as usable as possible. As discussed, there are many different styles of playing and thus a grammar which would encapsulate all of these styles would be difficult and not the intention of this project. However, unlike the Bol Processor, the application should be module-based allowing new or variations of the grammar to be used by the application in the generation.

# Chapter 3

# Requirements Document

The purpose of this project is to essentially two fold:

1. To develop a simple user-friendly system for generating Tabla patterns
2. To show that Tabla compositions can be modeled as a formal grammar composing as an intersection of various existing grammars

With these two main objectives in mind there are some issues which we must take into consideration illustrated by the literature review. The requirements specification will give a high level overview of what the system should deliver and to measure the success of the system against. The requirements analysis identifies the problem domain highlighted by the literature review ensuring that the areas discussed have been considered and dealt with. It has been decided that the primary goal of the project is to fulfill the second point listed above. This is because the main idea of the project is to show the grammatical modeling of the Tabla compositions before the presentation of the language in a user friendly format.

## 3.1 Requirements Elicitation

With these primary goals in mind it has become difficult in identifying the main user(s) of this application. Although the project intention is to develop an application it may not be considered strictly a 'software development' project as there is an element of experimentation in the main goal of this project. The author will be gaining further understanding of the Tabla compositions throughout the design and development of this project therefore causing the design and development subject to change. As a result, several elicitation methods have been adopted to gather

the necessary requirements to enable a sufficient execution of the project.

The first has been a brain-storming strategy. This is due to the lack of knowledge of the domain and therefore development of ideas need to be informally described which were gathered from the literature review. Although there is a lack of formalism to this technique, it encourages a rapid understanding of the domain [28]. In addition, there will be a requirement to consult those with a good understanding of the application domain. This will include the author's supervisor John Ffitch as well as other well known Tabla players such as David Courtney and Jim Kippen.

The next strategy enables the further elicitation of requirements for the design of the system after the first strategy. This was predominately for the design of the modularity of the system and explicit data structures that would be required to build the grammar. Therefore a UML technique known as sequence diagrams were used to elicit these requirements. A UML technique was sought for as it is a recognised tool for object-oriented applications. This is only one type of interaction diagram (the other being collaboration diagrams). However the sequence diagrams allow us to show the interaction and behavior of objects within the system rather than only concentrating on message passing between the objects. Only a subset of these diagrams have been developed as time has not permitted a full implementation of all diagrams.

Although limited, these methods have been adopted to gather the requirements to develop the intended system and allows scope for extension and change should they need to throughout the development process.

## 3.2 Definitions and Structure

Throughout this requirements specification there are some formal terms which require some definition and explanation.

- **Optional** If a requirement has this tag then the system should only implement if it is feasible in terms of time. It's implementation will extend the functionality or performance of the system.
- **Required** This is a requirement which the system must fulfill. These requirements will be used to check the success of the system against.

## 3.3 Constraints

We need to establish the main constraints on the project before creating the requirements specification as we will wish to confine the gathering and development of the requirements to the limitations imposed. The constraints help to manage the validation and verification of the requirements and to ensure the best way in which to develop the project. These constraints have come from numerous sources including the literature review, the final year project specification and discussions with the project supervisor.

1. Time - The project must be completed by 12pm on 12th May 2006. This will constrain the functionality and full development of the system and will limit the extent.
2. No Finances - The project should have no financial requirement. All resources and tools used in the project should be free and may influence design decisions and alter requirements.
3. One developer - As there is only one developer for this system and in conjunction with the first constraint, will again limit the scope and extent of the functionality.
4. Limited Domain Understanding - The literature review highlighted the fact that it will not be able to gain a full understanding of the application domain and thus a pragmatic approach will have to be taken in the selection and development of Tabla compositions.
5. Platform - access to different platforms are limited and therefore execution on multiple platforms may reduce the extent of cross platform ability.

We can now discuss issues raised in the literature review and relate them to the requirements that have been gathered from the brain-storming method. This analysis will then be succeeded by a brief list of high level requirements generalising the full specification which can be found in the appendix along with the sequence diagrams created for the requirements elicitation.

## 3.4 Analysis

### 3.4.1 Technology and Platform

The C programming language was discussed in the literature review as the main language to implement this system due to familiarity with the language and its flexibility and power. However, after further consideration particularly influenced by initial design ideas of data structures

and the user interface an object oriented language would be better suited. This is because using objects and the use of inheritance would provide a more natural way in which to develop the system. In addition, the potential size of the system is uncertain due to the experimental nature of this project an efficient or fast language would be beneficial to development such as C++ or Java. I have chosen C++ as it would contain all of the benefits of C as well as providing an object oriented framework. Although Java may be still a good language for developing a multi-platform application and is an object oriented language, performance could be slow and due to it's extensive library and various API's available could provide a complicated framework to work with.

There is the concern of memory management but I feel that the advantages of using a versatile language like C++ outweigh the disadvantages.

I will be using Visual Studio .NET as an IDE to develop the system. This is because the use of a familiar IDE will aid the rapid development of the system. The IDE's debugging tools and user-friendly interface increases efficiency of development along with large development support allows the simple integration of other APIs. Special care must be taken using this technology when developing a potentially cross-platform application. This is because the .NET framework will be available which will have to be avoided and only the standard libraries may be used - this will be dealt with more in the design and implementation section.

For the graphical user interface (GUI), there are several APIs that have been considered including WxWidgets and FLTK. Qt has been chosen to develop the GUI with as there is an open source version of the API which works well with C++, it is flexible and easy to use - equipped with a designer and a relatively simple object model allowing a shorter learning curve ideal for this development.

To develop a usable application a feature which will be discussed is the ability to output the compositions to the sound card. This is not a sound modeling issue, but developing an interface similar to TaalWizard application [15]. This does not need be a complex task but the ability to output sound/samples of the compositions will require further investigation if the standard C++ libraries do not offer such a feature.

### 3.4.2 Understanding Indian Classical Music and Tabla Compositions

As seen from the literature review, Tabla compositions are intricate and complicated forms of rhythmic patterns requiring years of expert tuition and devotion to training in order to gain the

skill to play and compose. The compositions that were analysed were examples of more simpler and theoretical versions. They can be used in a performance but may be considered as too basic. As this project is focusing more on the proof of concept of the grammar modeling of simple compositions for performance, complex compositions will be disregarded for now. Jim Kippen discusses the complications and perceptions of performance in [18] and highlights the change in mood of the performer in relation to the mood and feedback of the audience. I will not be taking these ideas into consideration as that is not what this project is intending to show. Therefore when we refer to 'performance' it means the main output of the compositions to the user.

The next issue to be considered is the suspension of an artistic model. As just previously discussed and in the literature review, if these compositions are to be developed and modeled grammatically there is a question as to whether the compositions are derived with an artistic influence or randomly. As the developer of this system, he does not claim to be a Tabla player nor claim to have artistic influences in Tabla compositions despite being a musician. This further supports the necessity of producing a set of compositions which work on a theoretical level. The most flexible way in which this should be implemented would be to formalise the compositions so that they can be developed to incorporate artistic influences if the system was to be extended.

The main elements to be taken away from the study of the compositions are two things:

1. Each composition must work in all Tals.
2. The structure and the phrasing (i.e. the rules) of each composition must be maintained.

Each composition is a particular module which defines a subset of the language and therefore adhere to various and unique rules. These rules must define the composition independent of the Tals.

It should be noted that as each of the compositions have a particular 'function' and their use within a performance must be correct. As described by Courtney [9] many of the compositions although can be defined by their structure are also defined by their function i.e where they are used in a performance. Therefore when each piece is considered for grammatical modeling, their use and purpose should be acknowledged to produce a coherent performance.

The compositions that will be initially considered are the ones which were discussed in the literature review: Theka, Kaida, Mukhada and Tihai. Further compositions will be considered if time permits. The reader should read the literature review to remind themselves of the way in

which these compositions work.

### 3.4.3 Grammar Based Modeling and Generation

The three main grammars that were discussed in the literature review were the transformational grammar, tree adjoining grammar and the phrase structural grammar. The Bol Processor grammar was discussed but its technique is different to how the author wishes to implement this grammar. The phrase structure grammar is the main element used in linguistics to construct a language. This should form the main basis of the entire language. The phrase structure creates the underlying skeleton and will specify how small sections or subsections of the compositions will be constructed and define the way in which each of the bols are connected together.

An interesting deduction can be made if we interpret the Tal and the phrase structure as being synonymous to each other. This is the first main connection to be identified between the structure of Tabla compositions and a formal grammar which can act as a starting point for the development of the final grammar. This phrase structure in the form of a grammar should be appended to the system easily working independent of compositions. In addition to this comment, we should recognise that the entire system should work in a modular fashion. This means that despite the formalisation of the Tal and the compositions, the system should not need refactoring if the grammar of the compositions should change.

After the phrase structure has been established, a lexical insertion process must take place whereby we have a process of linking 'words' of the language to the phrase structure. However the 'words' in the language we are attempting to model have no morphology or semantics. Thus, in tree adjoining grammar terminology, we cannot lexcalise trees without the semantics of the word. We also have no artistic stand-point to insert bols or lexicalise phrase structure trees thus determining the bols to be used must be taken by a different approach. The main idea is to avoid the word probability strategy adopted by the Bol Processor as a way in which to choose or select bols. Here is where the Tree adjoining grammar can offer its power of modeling all elements as trees. Since we are dealing with a tree structure, having phrase structures or trees already lexicalised with bols which considered as an accepted sequence can avoid this problem. The only transformations required here are the ones specified by formal TAG; substitution and adjoining. Therefore we do not need a lookup for every bol in a sequence, but to store trees of bols which preserve sequences creating a knowledge base of bols.

Once we have set up phrase structures which have been lexicalised, we can use transforma-

tional rules to manipulate the tree into compositions. The transformational rules therefore give the practical ability of creating the compositions from a base (i.e. a phrase structure) and here we can separate the theoretical creation of the compositions from the artistically influenced compositions.

The transformations can be fixed transformational rules which grouped together can perform a range of transformations and constitute a formal transformational grammar. This is keeping consistent to the way in which the transformational grammar is used for modeling linguistics. But as we have noted the similarity of the Tal and the phrase structure we could have a varying phrase structure for a single composition and thus creating transformational rules would be dependent on the Tal which we would want to avoid. More specifically, we would have to create a transformational rule for every Tal to perform the same task. Therefore we must make sure that the system does not have this dependency in order to have a modular based system and grammar. Therefore the implementation of the structural index for each transformation must be made so that it can generalise the phrase structure.

As Chomsky described the transformational grammar in his work Syntactic Structures, we would want to develop a way in which to make the transformational rules as simple as possible and as generic as possible to use it as a flexible tool. This would involve developing generic way of specifying the structural index - pattern matching the nodes of the tree and then performing the structural change upon the tree using the identifies nodes. This involves any of the four main families of transformations [14] substitution, permutation, null or insertion. However, as previously mentioned, there are no semantics behind the words of the language therefore the transformations must be based on some interpretation of the tree as a whole rather than the semantics.

### 3.4.4 Global Structure of Performances

Once we have created the individual compositions, we will need some sort of way in which to control the global structure of the performances. The pieces will have to be chosen to play in a certain order dependent on their function, along with a decision as to which piece would be played and at what intensity dependent on the mood of the player. The idea of the mood is to describe the intensity or style of the composition. Since we do not have an artistic stand-point we therefore have no sense of 'style' and therefore we shall define mood as the intensity of the performance.

This should reduce the use of random processes in the selection and creation of the compositions and instead moves more into the notion of using stochastic processes. From the literature review we identified compositions to be either cadential or cyclical. This will help the selection of compositions within a performance and tailor the compositions to perform their intended task. For example a Tihai is a cadential piece and therefore is used to end a section of or the entire performance.

### 3.4.5 Notation and Interface Issues

As Indian classical music has no notational system we must develop some method of representing the composition. The main notation that has been used in current literature is the phonetic spelling of the bols. These take on many different spellings though the most common will be used. The main reference for this is David Courtney's website.

Sometimes the spelling is specific to a particular composition such as the Kaida. Therefore as long as the rules are not breached for the compositions, alternative spellings could be used in some cases.

There will have to be some input by the user to kick start the generation. Therefore, the user could make a number of selections such as the Tal and the type of composition (solo tabla or accompanist), they would like to generate. This may then give the basic outline to create the performance. Other parameters which could tailor the performance should be considered.

The final aspect in terms of the interface which should be implemented if time permits is the ability to play the compositions using samples of the bols. This would enable the performances to be appreciated better and to gain a better understanding of the performances. The interface should provide the necessary controls to enable this functionality.

## 3.5 Requirements Specification

Here is a general outline of the functional and non-functional requirements for this system. These are high level requirements which the system must adhere to and are considered the most important or the required requirements. Please refer to the appendix for a full list of functional and non-functional requirements including the optional requirements.

### 3.5.1 Functional Requirements

#### Grammar Modeling and Transformations

1. The system should be able to create data structures to support the phrase structure of the language which models the Tal which the compositions rest upon. The phrase structure grammar should be easily created consisting of production rules and a start symbol which uses some method of creating a tree structure which the grammar defines.

2. The system should be able to have bols entered easily into the system and give specification as to what trees they can be combined as. These bols can then be adjoined to existing phrase structure trees in order to lexicalise them.

3. The system should be able to use lexicalised trees to perform transformations on, using transformational rules. The basic transformation function should be generic enough to specify the structural index and structural change independent of the specifics of the Tal. Thus the same transformation should not have to be manually created for every Tal. After an application of a transformation or a set of transformations should derive the intended composition.

4. Each composition should work modular to the system and an alteration in the grammar should not force changes to be made to the rest of the system.

#### Global Structure

5. The system should be able to provide some performance parameters for each composition which allows the tailoring of the composition to conform to the global structure of the performance.

6. The performance global structure should choose compositions appropriately using each one for its intended function/ purpose.

#### Graphical User Interface

7. The interface should allow the selection of the performance parameters which initially should be: Tal and composition type i.e. solo or accompanist. These will be used for now as high level parameters but may be refined or altered during development.

8. There should be a button which generates the entire performance and displays the generated composition in text format using the bol names as notation.

### 3.5.2 Non-Functional Requirements

**Performance, Extendability and Error Handling**

9. The system must not be slow or use up large amounts of memory. Tree structures may grow for large compositions but this should not take up adequate spaces of memory thus memory management must be taken care of within the system.

10. The system should be made so that it can be easily extended, particularly addition of Tals and compositions.

11. The system must handle errors internally and if any errors are caused by the user then they should be reported back to the user with a meaningful error message.

# Chapter 4

# Design and Implementation

## 4.1 Introduction

This section deals with the design of this system and discusses the implementation process. By describing and analysing the two discussions simultaneously, comparisons can be easily made along with reasoning behind implementation decisions. As this project has an element of experimentation, initial design decisions made will be explained as well as subsequent changes due to implementation issues or corrections to design will follow.

Initially a high level description of the main elements of the system will be given with an explanation of the purpose of each element. The next section will then start from the initial parts of the system and describe, much in the order that was outlined in the requirements specification section, how the system works from creating the basic phrase structure to the design and construction of the interface. Again, not all elements will be analysed in full detail, but the most fundamental objects, algorithms or the parts of the system which caused most difficulty will be discussed.

## 4.2 High Level Design Overview

With regards to the requirements specification and the literature review one of the main ideas that has been kept in mind throughout the design and development of this system is the system's modularity, constructed in such a way so that elements can be taken out or altered easily. The final architecture that was constructed for this system can be seen in Figure 4.1. This is not a

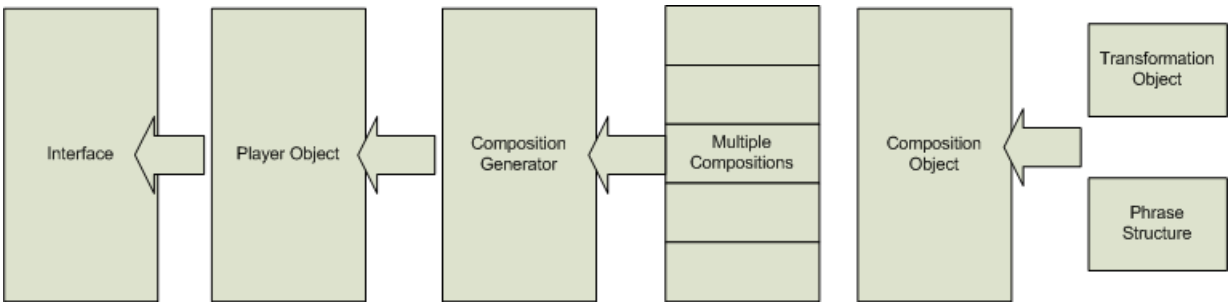


Figure 4.1: High Level architecture diagram of the system

class diagram despite the similarity between this diagram and the systems object model. This is more of a generalised version of the object model to outline the main elements in the system and to describe the dependencies between elements. Here is a description of each of the elements and the changes that were made to them to reach the final architecture as shown above.

### 4.2.1 Transformation Object

The Transformation Object holds all of the functionality which performs the main transformation to a tree data structure. The object holds one fundamental method: transformation() which takes primarily the structural index and the structural change as strings and a pointer to a position in the tree to begin from. As previously described, the transformation is a combination of four main functions: permutation, substitution, nulling and Insertion. The names essentially speak for themselves in terms of their purpose, though detail will be given on their semantics in the next section.

Although the transformation() method is only a tree manipulation procedure and could exist as a function on its own, it was unclear from the beginning how many different functions would be needed in order to perform a transformation and thus required it's own separate class. In addition to this, the requirements specified that there should not be single dependency on the Transformational Grammar and the system should be flexible enough to incorporate other methods. Therefore having the transformation function inside its own object seems sensible for modularity and flexibility.

### 4.2.2 Composition Object

This is the main parent class whereby all compositions that are created in this system must inherit from. In the original design of the compositions, the Composition Object did not exist and each of the compositions were created in their own separate classes utilising methods local to the class. The only connection between the compositions and the Transformation Object was that each composition instantiated the Transformation Object. This is still true but the instantiation occurs within the Composition Object instead.

There were two main reasons behind the birth of the Composition Object. Throughout development, the number of functions that was being replicated within each composition object was increasing and therefore it was made clear that the classes shared common functionality. Some of these functions varied slightly between compositions and as a result the functions' parameters and the algorithms were altered in order to make them more generic and moved to the Composition Object. An example of such a function would be changeLayakari(), however this method will be explained in detail later.

The second reason for the birth of the Composition Object was to have the ability to create the compositions in any Tal. Originally it was thought that the Tal structure would be created in the Composition Generator and thus remain independant of the composition objects. However as the phrase structure was synonymous of the Tal and to the defining composition the Theka, the phrase structure object could to be passed as a parameter to the composition. Instead, the Composition Object held the creation of the Theka composition which now serves the main way to create the phrase structure. By having this in the Composition Object each composition inherits the functionality to create the phrase structure in any Tal rather than being passed an object.

### 4.2.3 The Compositions

Each composition has been created in separate classes to keep modularity. The compositions that were chosen are as follows: Theka, Prakar, Mukhada, Kaida and Tihai. For now, although the Tihai is a seperate composition it exists as mainly part of the Kaida. The motives in choosing these compositions are what time permitted to implement and to get a fair number of cyclical and cadential pieces. As the Theka describes the Tal and excluding this (although it is considered a cyclical composition), we have two cyclical (Prakar, Kaida) and two cadential (Mukhada, Tihai) compositions.

The basic idea behind each of the compositions is that they use the phrase structure grammar to create the underlying tree structure, then use the transformation object to create transformational rules which can be applied to the tree to create the composition. Each composition can either dynamically create the transformational rules by analysing the tree at run-time, or can already contain fixed rules. To maintain consistency between the composition classes, they implement a single method which starts the generation therefore each class can be used in the same way. Therefore when each composition has been generated, there is a public property which is a pointer to a node. This will always indicate the beginning of the composition specific to that object. As a result, there is a uniform and consistent way in which the compositions can be referenced.

### 4.2.4 Composition Generator

This particular element is also a symbol for a single object in the system. This is the main object which holds compositions that exist and will have to be modified if new compositions were to be added to the system. Although its role has not altered throughout its creation, it's size and power has done. When the Composition Object was created, much of the functionality that existed in the Composition Generator moved to the Composition Object for reasons described previously. The role of the Composition Generator is to serve as an engine to deliver a composition in a specific Tal. Thus the parameters submitted to the Composition Generator are the composition name and the Tal. There is a third parameter which will be discussed later.

### 4.2.5 The Player Object

The player object is the main object that makes the decisions as to what will be in a performance. As previously discussed, there will be no artistic model which will influence individual compositions nor the performance. Therefore this object merely acts as a higher level object to simple make decisions as what to generate next. The player object simply asks the composition generator to generate three compositions for an entire performance. To give some dynamics to the performance the player object makes decisions (rather crudely for now) to generate a variation in the performance. This is not really an artistic influence but just to reflect a change in the intensity in the performance, i.e. the mood. The main model that has been adopted is that if the player is in a 'heavier' mood then the compositions longer and more vigorous. This could be argued as an artistic influence upon the compositions which the requirements has specified not to implement. However it was found that by having this as a property of the player object would dictate merely the size and intensity of the performance and not the style of the composition.

Originally the 'mood' was not considered as a parameter but it came about as a result of the compositions being too similar in size. By having the composition generator as an intermediary element between the player object and the compositions there was a place in which the players mood could be translated into affecting the generation of the compositions.

### 4.2.6 The Interface

There is not much to be said about the interface at this level. From the beginning of the design, the main aim of the interface was to provide a way in which the user could submit the overall performance parameters and to be able to read the composition text as a minimum. Therefore it was designed from the start that by having a Player Object, the interface would only have to instantiate one object passing the performance parameters submitted by the user to the player object only. This would create the entire performance and adhere to the MVC model.

## 4.3 Detailed Design and Implementation

A greater description of the design and implementation will now be given. The discussion will begin from the basic data structures in the system to the main objects of the system. Special dedication will be given to the compositions and to their semantics as they are the fundamental aspect of this system.

### 4.3.1 Creation of the Phrase Structure Grammar

As discussed in the literature review and requirements analysis, the connection has been made between the phrase structure of Tabla compositions and of the Tal implying the initial phrase structure for Indian classical rhythm. The compositions are essentially the specific use of bols defined using the criterion described by Courtney, but are 'equivalent' in any Tal.

This is quite a remarkable realisation as we can create a set of basic production rules which generate the structure which after some analysis reveals more interesting capabilities. The main composition which allows us to analyse the Tal is the Theka. As shown in the literature review this is the composition using an accepted sequence of bols to describe the number of beats in the Tal and to describe the clap/wave sequence. As an example we shall use Teental to see how the production rules are created and then the tree generated from the production rules, detailing the way in which has been done. The Theka for Teental can be found in Figure 4.2.

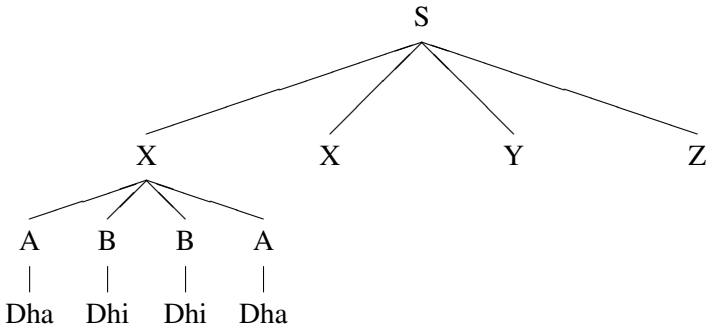

Figure 4.2: Theka in Teental

This Theka has 16 beats, constructed from four 'sentences' or vihbags consisting of four bols per sentence. From basic trial and error method the following set of production rules were constructed:

$S \rightarrow XXYZ$
$X \rightarrow ABBA$
$Y \rightarrow A(B)(B)(A)$
$Z \rightarrow (A)BBA$
$A \rightarrow Dha$
$B \rightarrow Dhi$
$(A) \rightarrow Ta$
$(B) \rightarrow Ti$

The production rules work by starting from the start symbol S and replacing the symbols on the right hand side with the other production rules with the same left hand side symbol. This is done repeatedly until no more rules can be applied. The bracketed characters indicate a relationship between the bracketed rule and the rule with the same letter such as B and (B). [18] described this as this as the Khula/Band homomorphism that mainly appears in the Kaida composition. However as there is a representation for it here as well - i.e. the 'band' bols are used for the 'wave' section of the Theka. Although whether creating this mapping here will be necessary, there seems to be no harm in creating the mapping between the two structures explicit and it actually indicates the underlying structure of the Theka independant of the bols.

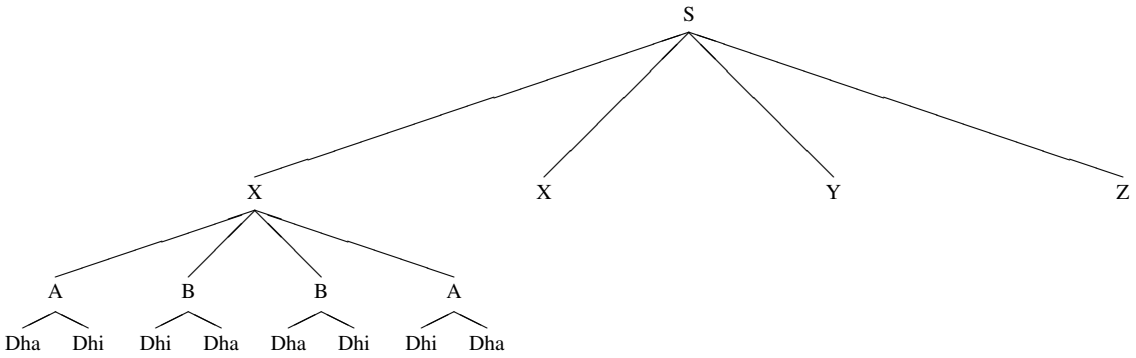
The final structure that is created by the application of the production rules is shown in the diagram below:




From the diagram only one branch has been fully expanded out due to space limitations. From this design, we can see that the creation of the Tal from the Theka can be easily constructed which yields this tree structure.

The tree is an N-ary tree consisting of node objects and pointers to other nodes. Each node is an object which primary has four main properties: a pointer to a parent node, a vector containing node pointers to children and two other properties which will be discussed shortly. A vector has been used as the structure is an N-ary tree and thus the number of children will vary. The vector object offered by the C++ library is versatile and allows easy manipulation as it can be treated as an array.

Before we move on to the discussion of the implementation of the phrase structure grammar and the creation of the tree, it may be a good idea to look closer at the tree and observe the information that it gives us. The 'S' character symbolises the start or root of the tree, and the progression of the composition moves depth first. The first set of children (X X Y Z) defines the number of sentences there are in the composition. The second set of children for each sentence (A B B A) defines the number of beats per sentence. This is great news for creating structures of compositions as it has defined not just the cycle of beats (16 beats, 10 beats etc.) but also how many bols we have per beat i.e. the layakari. For example if we were to create the Theka with two bols per beat (also known as Dugun) we would have this structure:




This has provided a powerful way in which to control the structure compositions by use of the Tal. This is a slightly different way in which to control the layakari than the way stated in the literature review section by using the tree adjoining grammar though the principal is the same. The tree adjoining grammar will become more useful in the 'Bag of Tricks', another part of the system which will be discussed later.

To ensure that we preserve the Theka's tree structure it was decided that for each node there be an associated 'type' of data type integer. This integer corresponds to the level of the tree - Start = 0, Sentence = 1, Beat = 2, Bol = 3. This is the third property of the node object. The final property is the value, which has been created as type string. This is the just the value that each node holds i.e. 'S', 'X', 'Dha'.

Another observation that can be made is the automatic lexicalisation of the tree. If the rules containing the bols are removed we are left with the skeleton of the Theka - a theoretical representation of a Tal in essence. However, by having the production rules to add the bols, we have lexicalised the tree producing the Theka. This is why using the Theka as a way to generate the Tal has worked so well. It was considered whether we could model this more closely to a natural language and design the phrase structures per composition based upon the semantics required for that particular composition and the transformational rules were made so that the tree would be manipulted to create the composition in a particular Tal. However, this creates two problems. Firstly creating an abstract phrase structure for each composition would entail a modeling of each composition based on semantics which is not clear nor understood from the authors understanding of the compositions. Secondly the lexical insertion process would have to be done manually, i.e. for a particular composition the bols would have to be chosen and appended to the phrase structure tree before the transformational rules were applied.

By using the Theka composition as a starting point, we build a skeleton of the Tal which is also a lexicalised tree. Then this tree is developed into a composition through the use of transformational rules. This is not only a better method for the reasons stated previously, it is the way in which the language of Tabla compositions are taught to a Tabla player. Initially the Theka is taught to gain an understanding of Tals. Then this composition is built upon to produce small variances such as Prakar and other compositions. Thus this method of having the phrase structure was decided.

Throughout implementation it was thought that although the leaf nodes serve the same purpose as the other nodes of the tree, the leaf nodes may require other properties, especially when it comes to the interface, such as assigning audio samples to specific bols. Therefore the object BolNode was created which inherits from the node object. This way the leaf nodes can still be stored as children of other nodes with the ability to add additional properties to the BolNode object.

In order to create the tree structure from the phrase structure rules, two data structures have been created. Firstly is the tree object. This object contains several methods used to create a parent node and add children nodes to the parent, (this includes containing overloaded functions for dealing with adding BolNodes and normal node objects). The next data structure that is used is the Phrase object. This is an object which will hold all of the rules for a particular phrase structure grammar. For every Theka, a single phrase object is instantiated. The intended design is to have one object that holds the rules and thus has a global property which stores pointers to all of the rules. Thus an addRule method has been created in the phrase object which takes four arguments: the left hand side symbol and the right hand side symbol(s) both as strings, and an associated type for each, both as integers. If there is more than one symbol they are represented as comma separated values. For example, the initial rule for Teental, $S \rightarrow XXYZ$ will be added to the phrase object as follows:

addRule(“S”,0,“X,X,Y,Z”,1);

The method parses the strings and then creates a tree object, making the left hand side symbol the root and then creating a node for each of the symbols on the right hand side, appending them as children to the root in the order the characters are passed. In addition the types of the nodes by the values passed in the parameters of the addRule method are set. The final stage in this method is the addition of the a pointer to the root node being appended to the global vector property of the phrase object which holds pointers to all of the rules.

Once this has been done for all rules, the single phrase object contains all rules for the grammar. By having the ability to add rules with their types in this way means that little analysis must be done by the developer to create the Theka and the phrase structure is easily created. Using this phrase object, the tree must now be constructed as this only defines the grammar for the phrase structure.

Essentially the process required to construct the tree structure is recursive. This is because if one observes the production rules, we have a recursive process whereby for every rule R1, find the rule R2 in the phrase object which corresponds to the R1's right hand side symbols and make R2's right hand side symbols children of R1's left hand side symbol. This algorithm can then be applied to every rule until no rules can be found and thus the creation of the tree can actually begin as the recursion unstacks. This therefore has turned into a recursive-descent phrase structure grammar.

In the design of this algorithm, it was not taken into account that once a corresponding rule was found, there were two node objects which represented the same element in the tree rather than one. For example, consider the two rules: $S \rightarrow XXYZ$ and $X \rightarrow ABBA$ . In the phrase object there is a tree for the first rule which contains a node object with the value 'X' which is a child node of a root node with the value 'S'. Another tree exists in the phrase object for the second rule where the root node has the value 'X'. When these are 'adjoined' a decision must be made between whether we make the root node of the second tree a child of the root node of the first tree or vice versa. Since computationally no difference is made, either way will work. What was not taken into consideration was that one of the nodes must be destroyed or else a memory leak occurs.

We now have been able to construct the main phrase structure for compositions to rest upon. As previously explained, this functionality resides in the Composition Object thus all compositions created which inherit this object has access to it. We shall shortly find how each composition uses this phrase structure to create the final compositions. Before doing so, the next element that should be considered is the Transformation Object. This is the main tool which is used to create transformational rules and applied to the phrase structure tree which creates the intended composition.

### 4.3.2 The Transformation Object

The concept behind this object has been fairly straightforward in terms of design. However several issues arose throughout implementation.

It was designed that as this object was to be used repeatedly, and therefore should be a simple way in which to specify the transformations upon a tree. Chomsky's idea of the transformation was considered and it may not be necessarily to have such a function to satisfy the full requirements of this system. The transformational function that has been described in a transformational grammar for natural language in [7] has been to pattern match against the phrase structure tree by identifying nodes in the structural index to nodes in the tree. However, as described in the literature review and by observing the way in which the phrase structure tree has been created for this system, specifying the structural index will vary between different Tals as it is based upon the Theka. It also works by recursing through an entire tree attempting to pattern match the nodes with the structural index. The transformation function for linguistics works in such a way which is too specific to a natural language which carries semantics throughout the tree. The purpose of the transformation function within this system is to indeed to manipulate the phrase structure tree based upon particular semantics (of the composition) but not of the tree itself since there are no semantics tied to the 'words' of this language, or in music in general! (dealing with individual elements in rhythm or melody). To put it more explicitly, we have a varying phrase structure for different Tals and only have layers of the tree which can be identified. This differs to the phrase structure tree of a natural language whereby the nodes may represent a noun or verb or some other definition tied to the word based upon its semantics. Therefore, it was decided to make the transformational function fairly elementary whereby complex transformations can be performed within compositions by amalgamating smaller transformations together.

The design of this algorithm precipitated in the object to having a single method, transformation() which took four parameters. The first two would be the structural index and the structural change both passed as strings, much in the same way as used in the creation of rules for a phrase object. The next parameter would be a pointer to the node which the transformation would start from and the final parameter would be whether the transformation would be working with a pattern matching ability or not.

To make the transformational function complete, the ability to have the pattern matching facility was designed but it was decided that this would be an optional facility. To be more explicit, the structural index would take on two forms. Either the structural index would identify the child

nodes from the starting node passed as a parameter in the following fashion: “1,2,3,4...”. Else if the direct pattern matching facility was enabled then the structural index could be specified as follows: “Dha, Dhi, Dhi, Dha” i.e. identifying nodes by their value property. Therefore the structural index section of the algorithm was dealt with by two functions. However throughout implementation it was realised that despite the power gained by the direct pattern matching feature it was questionable whether it was necessarily required? In some respects it was hard designing this algorithm without designing and developing the compositions first, yet they could not be implemented without the design of the transformation function first. Therefore the ability to directly pattern match actual values was suspended as although this would be what the usual transformation function would do from Chomsky’s idea, our previous argument regarding the varying phrase structure and uncertainty of which bols are used, allows us to suspend the usual implementations of the transformation() function. Therefore the structural index will only apply to children of a specific node which will be identified by the following syntax: “1,2,3,4...”. Should the pattern matching functionality be required, the ability to extend the algorithm is possible.

The main way in which the structural index works is to store the nodes identified and relevant information about them. This would include the node itself, its parent (all of the nodes identified will have the same parent) and their position within the sequence of other child nodes. This is important when finding different permutations of the nodes. To implement this, a simple struct has been created to store this information which is then held in a global property of the Transformation Object - SIVector. A pointer to each struct created is stored within this vector.

When this section was developed, allowing the structural change to perform the necessary manipulations became difficult to manage. This is because this function was attempting to permutate and manipulate the node pointers in a vector with the vector still having reference to them. Therefore once the structural index has been completed, there is no reason why the main root (from which the transformation takes place) should need to have reference to them as children. Therefore they can be cleared. Figure 4.3 is a diagram to make this clearer.

If structural index is “1,2,3,4”, we store these nodes in the SIVector and then can remove them as children of the root as shown in Figure 4.4.

We now have a clear working space within the vector to permutate the nodes or copy new nodes into.

Next is the structural change procedure. As previously discussed there are four basic transfor-

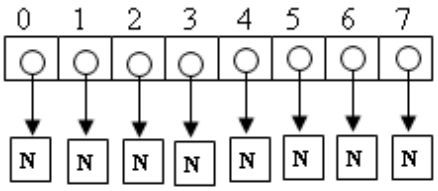



Figure 4.3: Vector of children nodes of root

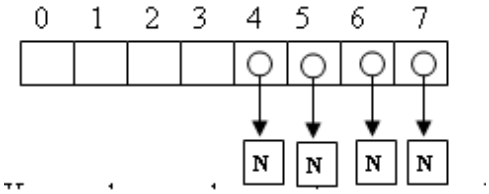



Figure 4.4: Vector of children nodes with identified nodes removed

mations which constitute a complete set of tree manipulations which can be used in any order: substitution, permutation, nulling and insertion, [14]. Using the structural index as a reference to the nodes, the structural change can be described as follows:

Given the Structural Index is: "1,2,3,4"
Substitution: "**1=Dha**,2,3,4"
Permutation: "**2,4,1,3**"
Nulling: "**-**,2,3,4"
Insertion: "1,2,**3+Ta**,4"

As it can be seen, the syntax used to describe the structural change is similar to that of Chomsky's transformational function. After the structural index algorithm we have reference to the identified nodes, and thus this section is a matter of parsing the structural change string and peforming the change. Using the similar delimiter function used in the structural index, it should be able to separate each element of the structural change string. Therefore we can iterate through each element and treat them individually. As we perform the necessary manipulations to the tree, we can then insert nodes back into the root node's vector of children at the appropriate place along with the changes required. It was found with the four main functions of the structural change that two of them were trivial to implement and two which were not so trivial. The trivial functions are the nulling and the substitution functions. This is because when we parse the element in the structural change and easily identify what needs to be substituted or nulled thereby

merely changing the value property of the node and then adding it back as a child of the root node - which is actually done by the permutation algorithm. The insertion and the permutation functions are slightly more complex.

The main concept that was kept in mind in the design of this algorithm was to keep the simplicity of functionality. To do so the structural change is initially broken down into separate elements. We can then check for the use of different operators used within each element which defines the function that is being performed, i.e. use of '=' operator for substitution or use of '+' for insertion. Thus we can create a function which will identify the operators and then perform the relevant function to manipulate the node/tree. Therefore here is the outline of the structural change algorithm:

```
For every element in the structural change string
{
find the operator(s) used in the element
if substitution operator is used{
substituteFunction(element_in_SC_string, positionWithin_SC_string)}

if insert operator is used{
insertFunction(element_in_SC_string, positionWithin_SC_string)}

if null operator is used{
NullFunction(element_in_SC_string, positionWithin_SC_string)}

Permutatefunciton(element_in_SC_string, positionWithin_SC_string)
}
```

Each of the functions require the actual element from the structural change string and the position of the element within the structural change string to be passed as parameters. This is so each function knows where in the structural change the element exists and the main vector of child node pointers belonging to the root it is dealing with as they are synchronised. This will be made clearer in a moment.

Originally this algorithm was designed based upon the example transformations shown above and thus the practical implementations of this altered the original design of the algorithm. This is mainly due to the practical implications caused by the two more complex functions of insertion and permutation. As previously explained, the substitution and nulling functions are merely changing the value property of the node and then inserting back into the vector of children. Therefore we shall now discuss the design of the permutation and insertion functions and then the effect it made on the general algorithm.

The permutation algorithm is quite a simple one but slightly confusing. Let us define the vector which holds the structs of the nodes of the structural index as the SIVector and the separated elements of the structural change again stored in a vector, SCVector. Finally the vector of the node pointers to the children belonging to the root as Children. We shall also let 'index' be a variable which is used as the reference to where we are in the iteration of the structural index elements. Then we can define the permutation as:

```
Children[index] = SIVector[SCVector[index].nodeUsed()]
```

The nodeUsed method is a pseudo function which retrieves the value of the node being manipulated. i.e. if the structural change is: "2,1,3,4" and index = 0, then nodeUsed returns 2. This means that the node referenced as 2 in the structural index is to be stored in the vector referenced as 0. Therefore as we iterate through the structural change elements we are also synchronising the iteration through the Children vector, i.e. whereever we are indexed in the structural change vector we are referencing the same place in the root nodes vector of children.

The next function we must deal with is the insertion function. The basic ability of this function is to insert new children into the root's vector at the appropriate position. This means either side of a particular bol, e.g. "Dha + 2 + Dhi". If this is embedded amongst the other elements in the structural change then this becomes slightly difficult to manage, e.g. "1, Dha + 2 + Dhi ,3,4", although only the element "Dha + 2 + Dhi" is passed to the insert function.

Essentially what we must ensure is the correct order in the insertion of the new nodes. Also, the structural change specifies the value property of the new nodes that should be inserted. When new nodes are inserted they are usually bound to an existing node using the '+' operator to indicate which side of the node the insertion is taking place. This method could have many interpretations to produce the same effect for example, using the previous example "1, Dha + 2 + Dhi ,3,4" could be written as: "1+Dha,2, Dhi+3,4". Either way is possible with the following method:

```
Find the value of node we are referring to, e.g '2' in the above example

Add elements to the left of it - if it is a string then a node must be created for it
and then insert it to the children vector of the root

Add the node referenced as '2' from the structural index
```

```
Add the elements to the right of it - if it is a string then a node must be created for
it and then insert it to the children vector of the root
```

However, due to some limitations of the Vector object in C++, some problems have arisen with this method. The first is as we are inserting new child elements. New elements are added to the vector by either the push() method which appends to the end of the vector or if the push() method has already been invoked, then that space can be replaced. Thus in the above transformation example: "1, Dha + 2 + Dhi ,3,4", the insertion of the nodes containing Dha and Dhi are actually spaces in the vector indexed by 1 and 3 respectively. Thus we create a new vector which will hold the entire vector of node pointers which will replace the vector in the root node. Therefore if the structural change is: "1, Dha + 2 + Dhi ,3,4" the above algorithm works as follows:

```
Find the value of node we are referring to, e.g '2' in the above example

Iterate through the entire vector of the root node's children

if the node is the element we are dealing with the, e.g. '2'

Add elements to the left of it - if it is a string then a node must be created for it
and then insert it to the newVector

Add the node referenced as '2' from the structural index

Add the elements to the right of it - if it is a string then a node must be created for
it and then insert it to the newVector

Else append to newVector

Replace the root's vector of children with newVector
```

This allows us to insert as many elements on one side of a node as the other. However, the second problem that arises due to the limitation of the Vector object is the fact that since new nodes exist, they interrupt the original elements identified by the structural change string. If we look back at the original structural change algorithm we find that it iterates through the elements of the structural change string. Thus, with the introduction of the new nodes to the right of the node we are inserting around comes into the way. Figure 4.5 illustrates this diagramatically.

It can be seen from this diagram that when the loop moves to the next item, we have found a node which we are not meant to consider. Therefore, at the end of the insert function the number of elements right of the considered node is returned which is used to increment the position of the permutation and the subsequent transformations.

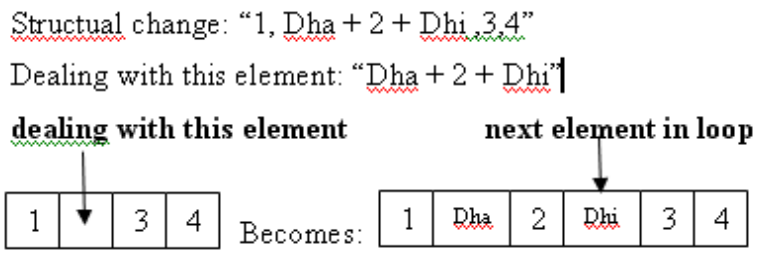


Figure 4.5: Problem with inserting new nodes

This completes the rather extensive algorithm of the transformation function. We now have the ability to pass the structural index and the structural change with the given syntax and pass the starting node which the transformation will begin from. The transformation does not make a copy of the tree that is passed - the transformations that take place are upon the tree that is passed i.e. pass by reference. The management of which trees are to be passed to the transformation function is not the responsibility of the transformation function.

Now we have looked at the transformation function, we can now begin to see how it is used by compositions to develop the transformational rules. To begin with, one of the simplest compositions will be dealt with: Prakar as this is an extension to the Theka and thus uses very basic transformational rules to develop the composition. We shall then progress to the most complicated composition - the Kaida, which also incorporates the Tihai composition.

### 4.3.3 The Compostitions: Prakar and Mukhada

The Prakar composition bares a strong relation to the Theka. The Theka, although a composition used in performance can be viewed as dull or mundane when heard or played. However it's functionality, particularly when used as a way in which to maintain the Tal (when a Tabla is used as an accompanist to another instrument), is extremely important. To remove the mundane sound of the Theka, slight variations of it are played which produces the Prakar composition.

Here is the Theka and a Prakar composition both in Teental (the comma denotes the next sentence):

**Theka:**
$| \; Dha \; | \; Dhi \; | \; Dhi \; | \; Dha$

| $Dha$ | $Dhi$ | $Dhi$ | $Dha$
| $Dha$ | $Ti$ | $Ti$ | $Ta$
| $Ta$ | $Dhi$ | $Dhi$ | $Dha$

**Prakar:**
| $DhaDha$ | $Dhi$ | $Dhi$ | $Dha$
| $DhaDha$ | $Dhi$ | $Dhi$ | $Dha$
| $DhaDha$ | $Ti$ | $Ti$ | $Ta$
| $TaTa$ | $Dhi$ | $Dhi$ | $TiRaKiTa$

It can be seen that the Theka provides the skeleton upon which the variations can take place to perform the Prakar. It is the variations which are spoken of that describe the transformational rules to generate the Prakar. After some observational analysis of Prakars in numerous Tals, here are three strategies for the transformational rules that can be used:

1. The bol or set of bols of the first beat of each sentence can be duplicated
2. The bol of set of bols of the last beat of each sentence can be duplicated
3. The first/last beat of the last sentence can be replaced by a set of accepted bols

These strategies are fairly primitive ways in which to produce Prakars, though in keeping with the constraints of the requirements, we are only going to deal with simple compositions. These strategies are also defined for any Tal making them very versatile.

Therefore to implement them we can have a method per strategy which takes a node pointer as a parameter. For the first two strategies the algorithms are relatively simple, especially since we have a tree structure which defines the sentences and beats per sentence. Both methods identify the appropriate beat in the sentence by traversing the tree and calculating which branch to select. They then perform the following transformation:

```
transformation("1","1+1",rootNode,false)
```

This transformation essentially identifies the first child node of rootNode and inserts a duplicate of it. The fourth parameter as explained before sets whether the transformation is using a direct pattern matching method for the structural index which this is not. I.e. we are not identifying

the nodes by their type or value, only by the ordering of the nodes.

The third strategy for generating the Prakar uses a transformation which is not directly related to the transformation() function object but more to tree adjoining transformations. This is because rather than manipulating a tree using the transformation() function, the desire here is to replace a part of a tree with another tree which the transformation was not specifically designed for. It is possible to use the transformation() function to perform this particular transformation, however it was considered easier and simpler to implement if there was another function to handle this. Therefore as we have the Transformation Object, we can implement this new transformation in the same object but as another method.

One of the ideas that is now adopted is taken from the Bol Processor. This was explored in the literature review which looked into how bols can be grouped together. For example having: Ti Ra Ki Ta or Ta - Ti Ta Ki Ra Na Ka are considered acceptable sequence of bols. We can have a tree structure which can hold these bols but it is not vital as there is no definition to the connection of the sequence of bols to a Tal with a specific layakari. What must be preserved is the ordering. Therefore the storage of these bols does not have to use a tree structure though we will see that by using a tree structure, it will enable a simpler way in which to create these sequences within the system.

The sequence of bols can be used in virtually any Tal and layakari. Courtney describes them as having a 'Bag of Tricks' for when improvising [9]. This is useful as we can create a store of these sequences and mimic the way in which a real Tabla player may improvise and use them.

Therefore we need to be able to have a transformation which allows the appending of the bols appropriately. Therefore a small set of functions have been developed to perform these fixed transformations. The importance of these transformations in relation to the transformational grammar will be discussed in the evaluation section. However, now we shall show the design and purpose of some of these functions. Here is the design of a function named AppendBolsToTreeEnd(). This takes a vector of bols, the root from which to perform the transformation on, and the layakari i.e. the number of bols to be appended per beat. The role of this function is to start from the last beat of the last sentence from the root, and to append the sequence of bols starting from the end to the end of the tree and work backwards. This function has been implemented as this ensures that despite the length of the sequence of bols, it finishes at the end of the tree the bols are being appended to.

**The Bag Of Tricks**

This is actually a method in the Composition Object which stores each of the 'tricks' and can be easily extended by adding further tricks to it. This is done by creating a phrase object whereby each of the bols are added as rules and then using the recursive descent phrase structure function, adjoin the rules together to form a tree. The function BagOfTricks takes an integer as a parameter and returns the root of the tree which holds the sequence of bols. The integer parameter corresponds to a number which identifies the particular trick. The sequence of bols are arranged in order of increasing number of bols within the sequence. Therefore the trick with the least amount of bols has the value 0 assigned. The reason why the tricks have been arranged in such an order will become apparent when we discuss the use of stochastic processes and choices.

This method is used as part of the third strategy for the Prakar compostion. A sequence of bols is chosen from the bag of tricks and then appended to the Theka tree using the AppendBolsToTreeEnd. There is also a fourth strategy which uses a linear stochastic algorithm to choose two of the other strategies and apply them succusseively. Again, the actual details of stochastic processes will be discussed when we talk about choices.

The implementation of the Mukhada composition is very much like the third strategy of the Prakar. This is because the sequence of bols are usually small cadential type compositions. But rather than being compositions, they are merely 'licks' that one could use. However they are a good way in which tension can be built up by increasing bol density which ends on a Sam - the main philosophy of a cadential composition. In addition to the use of the bag of tricks, it has been said that there are a variety of sequences that could be used to constitute a Mukhada, [9] and [11]. For example, the bols of a Kaida theme could be used as a way in which tension can be built up. Therefore the creation of the Mukhada is like the Prakar whereby the root node of a Theka or a particular tree is passed and there is one of two strategies which are used: either a sequence of bols from the bag of tricks or a theme of a Kaida are used to replace a section of the tree. Whether this has the constraint of choosing a Kaida theme specific to the Tal of the composition it is transforming is unknown. Essentially it should not matter as we are just interested in the bols themselves rather than the rhythmical structure they are bound to. However since this issue is unknown, the function has been designed to use only Kaida themes which resides in the Tal the Theka or tree is in.

### 4.3.4 Choosing strategies and Stochastic processes

The Prakar and Mukhada compositions can use stochastic processes in order to choose a strategy and affect the variances in the compositions. If we are able to categorise and separate out each of the styles which develop the compositions, we are free in choosing which style we want. While the Prakar composition is a cyclical piece and the Mukhada is a cadential piece their use of stochastic processes differ but the fundamental idea behind the method remains constant.

The one idea which we can retain from a performance is the idea of creating tension by increasing the number of bols (providing they are a valid sequence of the language). In a cyclical piece we can use this method to fluctuate and vary the tension of the composition. However with a cadential piece after tension has been built up it should be release on the Sam.

Thus from a basic analysis of performances we can identify that the compositions used are played at greater length to increase tension. This can be interpreted as a linear stochastic process. Therefore if the styles and techniques are ordered into increasing tension then we can apply a linear stochastic process (negative correlation) upon this. In other words, if we request a strategy from a set of strategies, the stochastic process will have a higher probability of choosing ones which have a lower number of bols.

We do not want to go in greater depth into the analysis of performance as this is not the intention of this project. We merely would like the system to use a basic intelligent way in which to create a composition. Essentially we want to use a stochastic process in choosing the transformational rules.

The two main stochastic processes that were considered were the linear distribution and the triangular distribution, [6]. Diagrams of these processes can be found in Figure 4.6.

Although it could be argued that the increasing tension (i.e. increasing the number of bols) and the selection of the strategies could be modeled as a triangular distribution it seems more intuitive to use a linear distribution. We can be have a function which generates number between 0 and 1 and maps this between 0 and a maximum limit - a value which can be passed in as a parameter to this function. The limit is used as the number of strategies there are and then when a value is generated, it maps to a particular strategy. Provided that the strategies are ordered by increasing bol density then this should give a better and more natural way in which to select the strategies.

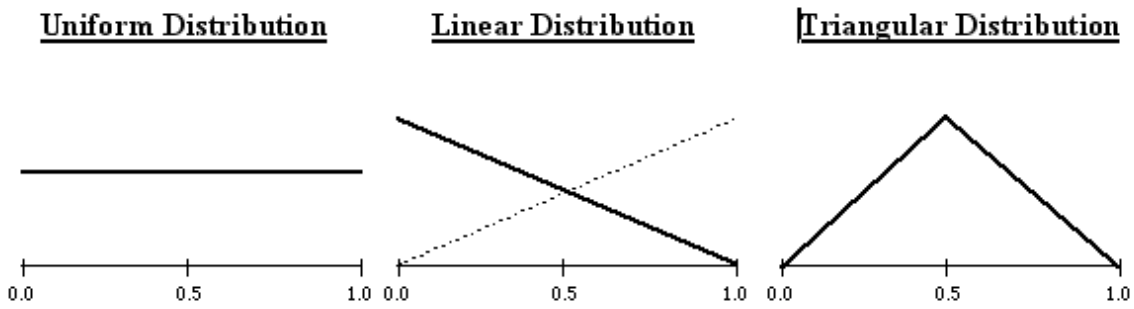



Figure 4.6: Stochastic Processes

As a result, the Prakar composition can use this method of selecting different strategies by using the stochastic algorithm to generate a strategy. On greater use of this method will reflect strategies being chosen generating increasing bol density of the composition.

### 4.3.5 The Compostitions: Kaida and Tihai

The most complicated composition that has been designed and implemented has been the Kaida. The main reason for this is because the understanding of the Kaida has continued throughout the design and implementation of this composition.

To begin with, lets reminds ourselves of the structure of the Kaida:

1. Theka
2. Kaida Theme
3. Kaida Theme in double the layakari of the theme
4. Variations
5. Bharan - optional
6. Tihai

Since there is a rigid structure, having a central driver method for the generation of the entire composition would be the simplest way in which to manage each of the sections.

Generation of the Theka is straightforward and we have dealt with this composition. The next

stage is designing a way in which to create the theme. Three things were soon realised about the Kaida theme during the design stages of this composition.

- The first is that there are different themes for each Tal (though these themes could cross into different Tals, but the specifics of this were not explained well enough in literature and thus has not been considered).
- The next thing was that the themes are not generated by the player. They are usually fixed, or based upon fixed versions much like the Theka. Thus it is not up to the Tabla player to make-up or improvise a theme themselves.
- Finally it was also realised that the theme defined no strict relation to the Tal. This meant that the layakari that is used when the theme is presented in a performance of a Kaida is up to the player.

When these issues were understood, it aided the design of the functions.

### Creation and Generation of the Kaida Theme

Since only the bols themselves of the theme must only be taken into consideration, we can insert the bols one at a time. This does not need any analysis by the developer or whoever wishes to extend this 'knowledge base' of Kaida themes. The only information that needs to be stored in the knowledge base of themes is the Tal that each theme belongs to. However, if the insertion of the bols can be carried out individually, we can create a phrase object much in the same way in which the phrase object is created for a Theka to compact the insertion of the theme. For example, the theme: **Dha Dha Ti Ta, Dha Dha Tu Na, Ta Ta Ti Ta, Dha Dha Dhi Na** can be be reconstructed by using a small phrase structure grammar using the following production rules:

$S \rightarrow WXYZ$
$W \rightarrow AABC$
$X \rightarrow AADE$
$Y \rightarrow CCBC$
$Z \rightarrow AAFE$
$A \rightarrow Dha$
$B \rightarrow Ti$
$C \rightarrow Ta$
$D \rightarrow Tu$

$E \rightarrow Na$
$F \rightarrow Dhi$

This is clearly more concise than inserting each bol individually, though it is not necessary since we do not have to presever the number of bols per beat as there is no definition to how the theme connects to the Tal. However to perform the transformations, the bols must be appended to some tree - i.e. with a defined layakari. Since this method is left up to the player the basic method is to calculate the lowest number of bols per beat in which the theme can be appended to a single cycle of the Tal. This is done by finding the number of bols there are in the theme and dividing it by the number of beats there are in the Tal. For example, the theme shown above can be appended to a Tal tree with one bol per beat as there are 16 bols and Teental has 16 beats, thus

(no of bols in theme/No of beats in Tal) = (16/16) = 1 bol per beat

In order to create a tree structure to append the bols to the Tal, the Theka composition was modified. The modification allows the creation of the Theka without the bols. This is done by a simple boolean parameter 'withBols' that is passed to the function set to 'true' to include the bols and false to leave the bol nodes off.

Now we need a function which will append the theme of the bols to a tree with the calculated layakari. The function AppendThemeOfBols() does this. The basic semantics of the function is to iterate through the bols of the theme and to append bolNode objects to each of the beats of the Tal with the number of bols per beat equivalent to the layakari. The function finally returns the root of the tree.

Now we have the ability to create the theme and present it in a phrase structure tree appropriately. The next stage is to create the same theme at double the layakari of the theme. Having this technique is quite common in Tabla compositions. If the layakari is 1 (i.e. one bol per beat), then this is usually known as Thah. If the layakari is 2 it is known as Dugun, 3 is known as Tigun and 4 is known as Chaugun.

**Kaida At Full Speed**

Therefore a separate function has been implemented which performs this functionality: ChangeLayakari(). The main idea is that it takes a pointer node to the root of a tree, the Tal of the com-

position and the desired layakari. Similar to the AppendThemeOfBols() function, we collect the bols off the tree that has been passed to the function and then we can append them back to the tree with the specified layakari. This requires the management of looping through the tree and through the bols to be appended.

What was overlooked in the design of this function was if the desired layakari is greater than the layakari of the original composition then the function will reach the end of the bols to append before reaching the end of the tree leaving a partially leafless tree. Therefore there are two decisions that can be made here: either begin the loop again and continue to add the bols until the function has appended to the last beat in the tree. Or to stop and erase the remaining leafless branches and just keep the partially filled tree. Since both functionalities seemed useful, the decision was to parameterise the function whereby either strategy is specified.

To change the theme into a full speed theme we want to repeat appending the bols until the tree is complete as this is what the rules of the Kaida dictate. The function FullSpeedKaida() calculates the normal layakari of the theme and then uses the ChangeLayakari() method to return a tree which has the theme at twice the original layakari.

### Creating the variations

Up until now the Kaida has gone through a number of parts and has introduced the development of particular functions and in some respects, transformations which are not the transformations predicted by the original design of the transformational grammar. We can now deal with the main part of the Kaida which are the variations. In order to create valid variations, further analysis of the variations was required.

The variations must follow these rules:

- Must use the bols of the theme
- Must begin with the Dora variation
- Must exhibit the Bhari/Khali structure (or the use of Khula/Band bols).

In order to manage all three rules, the following observation has been made with Kaida Themes; if we were to divide the theme into four sections we find the first two sections show the Bhari and the last two sections shows the Khali. This means the last two sections use the 'Band' bols.

The reader should re-read the literature review section to familiarise themselves with Band and Khula bols. However, for a quick revision, here are the Khali and Bhari sections of the Kaida and the Dora variation:

**Bhari:**
| $Dha$ | $Dha$ | $Ti$ | $Ta$
| $Dha$ | $Dha$ | $Tu$ | $Na$

**Khali:**
| $Ta$ | $Ta$ | $Ti$ | $Ta$
| $Dha$ | $Dha$ | $Dhi$ | $Na$

The Khali section re-introduces the Bayan bols again at some point in the fourth sentence though when this occurs is undefined. Thus when there is a variation there is always this duality of having the variation in the Bhari reflected in the Khali by using the Band bols. The Dora - a fixed variation which must occur first is as follows:

**Bhari:**
| $DhaDha$ | $TiTa$ | $DhaDha$ | $TiTa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$

**Khali:**
| $TaTa$ | $TiTa$ | $TaTa$ | $TiTa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$

The number of sentences has doubled as we are performing at double the layakari of the original theme.

Courtney denotes this sequence as: AAAB - AAAB, whereby A and B are assigned to the first and second sentences of both the Bhari and Khali sections. This generalisation of the Kaida Theme works with almost all themes in any Tal. The above Kaida theme naturally splits into four sentences as it is in Teental, but here is an example in Jhaptal - 10 beats:

**Kaida Theme:**

$| Dhin | Ghir | Nag | Ti | Te$
$| Dhin | Ghir | Nag | Tir | Kit$
$| Tin | Kir | Nak | Ti | Te$
$| Dhin | Ghir | Nag | Tir | Kit$

**Dora Variation:**
$| DhinGhir | NagTi | TeDhin | GhirNag | TiTe$
$| DhinGhir | NagTi | TeDhin | GhirNag | TirKit$
$| TinKir | NakTi | TeTin | KirNak | TiTe$
$| TinKir | NakTi | TeDhin | GhirNag | TirKit$

We can now model this by appending the bols of a theme to a tree of four sentences, whereby the first two sentences are the Bhari and the last two are the Khali.

Using this tree structure which we can perform valid transformations upon it. If we were to create the transformational rule for the Dora, then the structural index would be: "1,2,3,4" and the structural change would be for the Bhari section: "1,1,1,2" and for the Khali section: "3,3,3,4". Therefore to perform the entire transformation a function is created for creating a 2 copies of the original tree, perform the required transformations upon them and then appending the final sequence of bols to a Tal tree at the appropriate layakari - i.e. equivalent to the layakari of the theme at full speed. The DoVariation() method does this very task.

The subsequent variations are random permutations of the four sentences of the theme. Thus a random number generator is used to create a variation of the two characters "1" and "2" such as "1,2,1,2" or "1,1,2,2" which is done by a method PaltaString(). Then another function is used to find the corresponding numbers which constitute the Khali sentences, FindKhaliString(). Thus if the Bhari structural change is "1,2,1,2" the corresponding Khali structural change would be: "3,4,3,4". We therefore have the freedom to generate the initial Bhari section, and then run a method to find the corresponding Khali structural change. This allows strict synchronisation between the variation of the Bhari and Khali sections.

It must also be pointed out that when the Bhari structural change string is generated it ends with the last sentence of the Bhari. This a rule that the Kaida dictates and thus this can be hard-coded

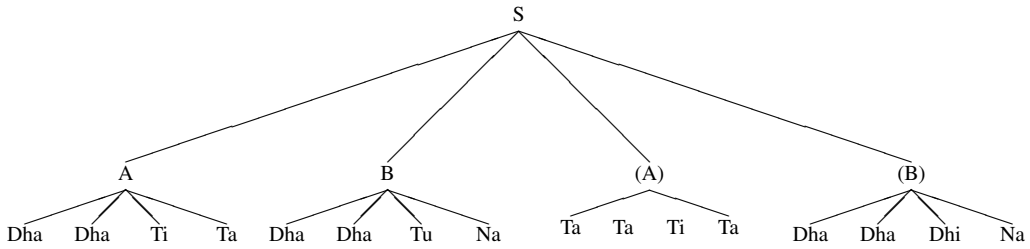



Figure 4.7: The phrase structure of the Kaida theme

into PaltaString(). Thus here is an example of a DoVariation command:

**Dora Variation:**

```
DoVariation(root,"1,2,3,4","1,2,2,2","3,4,4,4",TheTalOfTheKaida)
```

**Next Variation:**

```
DoVariation(root,"1,2,3,4","1,2,1,2","3,4,3,4",TheTalOfTheKaida)
```

**Further Variations**

The next set of variations are based upon the same tree structure but with some extra features. This is because these variations are more complex as sections of the four sentences are broken down into sub sentences and the variations are carried out upon these. The composition is still bound by the rule that if the subsentences are used, the variations must still exhibit the Bhari/Khali structure. Therefore to ensure that this is done correctly we can use the existing structure of the theme to first break down the Bhari section and then perform the same breakdown of the Khali section which should allow synchronization of the two sections as before. Figure 4.7 shows a diagram of the tree.

We choose a random sub-sentence from the Bhari section whilst maintaining the ordering of the bols and find the corresponding bols in the Khali section. The remaining bols of what was not chosen is separated into another tree. These new sentences are appended to the main Kaida tree as shown in Figure 4.8.

As with the previous variations, we must generate a valid string which will perform the structural change for both the Bhari and the Khali section i.e. using the DoVariation() command.

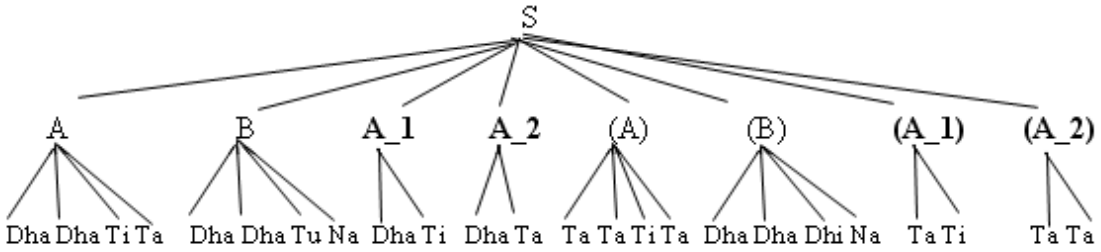



Figure 4.8: Kaida Phrase Structure with appended sub-sentences

The structual index is now given by: “1,2,3,4,5,6,7,8” as we now have 8 child nodes whereby the Bhari section are nodes 1-4 while the Khali section are the nodes 5-8. We can find a valid permutation of these nodes to generate a variation, bearing in mind that the number of bols that is used does not exceed a single cycle of a Tal as the Kaida Theme does not exceed this limit. Again, we are bounded by the rule that the permutation must end with the last sentence of the Bhari section i.e. node 2. We perform the task of finding the corresponding Khali string which ranges 5-8 by using the same method as previously stated by finding the corresponding values. Therefore, for a Bhari structural change string of “4,2,1,2” the Khali string would be: “8,6,5,6”.

This method is repeated a number of times depending on the request, and new sub sentences can be created, generating new permutations and variations. Figures 4.9 show this for the permutation: “2,3,4,2,2,-,-,-” and “6,7,8,6,6,-,-,-” given the structural index as “1,2,3,4,5,6,7,8”.

We can also use these trees to incorporate other techniques such as the use of rests. These are usually introduced as ways in which to vary the rhythmic patterns of a composition. Thus when the trees are generated, we can use substitution transformations which replace the bols with a pause or silent symbol, e.g. S. Thus it has been hard-coded that in the last variation, there will be a pause (in both the Bhari and Khali section) but randomly placed. The Kaida does not really dictate where such a pause will occur and thus the use of a random placement of it seems feasible.

The final part of the Kaida is the generation of the Tihai. The main concept behind this composition is that it is a repetition of three phrases of bols and ends on the Sam. The main issue behind this composition is the variety of the structure that it can take. This is because the mathematics of the beats that construct the Tihai varies dependant on the presence of other compositions. This is also contributed by the artistic influences of the Tabla player deciding on the variance of the Tihai.

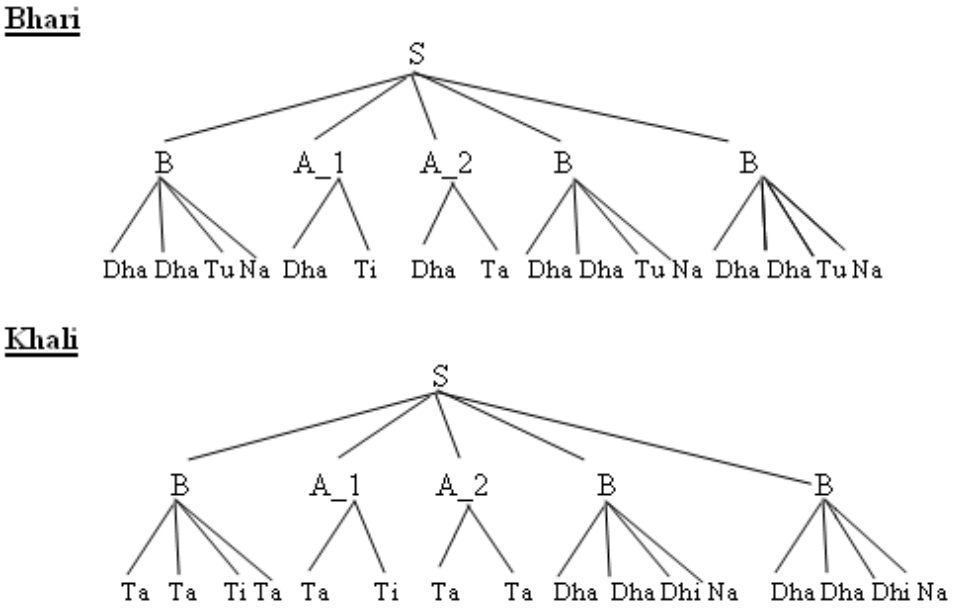



Figure 4.9: Resultant trees of variations

Therefore as we are choosing theoretical compositions the following methodology has been adopted which precipitates an algorithm quite naturally.

**What we know of calculating the Tihai**

As explored in the literature review there are two main equations that calculates the number of bols required per unit of a Tihai. The first considers a Tihai with no pauses (Dam). This is given by:

*((Number of beats to fill * layakari) + 1)/3 = number units in the palla*

With pauses separating the pallas:

*3*(number units in the palla) + 2*(number of units in Dam) = (Number of beats to fill * layakari) + 1*

In addition to this, the Tihai is sometimes prefixed by a small composition known as a Bharan which is a repetition of the Kaida Theme.

It is desired that we take each case at a time and attempt to develop the Tihai with those constraints. If the development of this composition is not possible then a variation will be made and the composition will be developed again. We also know that the Tihai uses the same bols of the Kaida theme thus the tree holding the theme can be used.

Using these rules we can generate a Tihai composition by first calculating the number of bols that is required for each palta (single section of the Tihai) for a Dam Tihai first and then selecting a subset of the Kaida them, enumerating the number of bols.

Since the use of the pauses and whether to include the Bharan are artistic choices, the strategy that has been adopted is to calculate a normal Tihai (i.e. a Dam) for a single Tal and if this is not possible then to introduce pauses to make the mathematics work out.

The algorithm essentially calculates the number of units within a Palta (NoU) and randomly selects nodes from the Kaida theme tree and enumerates the bols. If the number of bols exceeds the NoU then the selection process begins again. If this continually fails - i.e. there is no permutation of the bols which allow the NoU value to be reached, then the NoU is calculated as a Bedum Tihai and introduces pauses. The number of units in a pause begins as 1 and increases until the bols selected enumerate to the NoU value.

However the use of the Bharan can complicate issues as if the Kaida Theme performed at full speed only fills half of the Tal often the second half of the Tal is used as a Tihai if it is mathematically possible. Therefore the creation of the Bharan was parameterised to the generation of the Tihai. If there is a Bharan, the method would initially calculate whether the Bharan would fill a whole Tal or not. If not, then the remaining beats of the Tal are used as the 'Number of beats to fill' variable for the Tihai equations. Else, if the Bharan does fill a whole cycle of a Tal, the generation can continue as before.

This allows the flexibility in the generation of the Tihai from the Kaida Theme phrase structure but as opposed to transforming the tree, an unlexicalised Theka tree is generated and the selected bols for the Tihai taken from the Kaida Theme are appened to this tree.

This completes the design and implementation of the compositions. We can now talk about the final aspect of the main model of the system which drives the creation of the compositions. This is the Composition Generator and the Player Object.

### 4.3.6 The Composition Generator and the Player Object

Throughout the development of the compositions, the Composition Generator has always existed. After each composition class was developed they were appended to the Composition Generator object and thus its design and role is simple.

The object acts as an engine which will return a composition on request, i.e. it will instantiate the requested composition class. Having an engine to generate compositions allows a single object to consolidate all of the compositions and thus as we create new compositions, there is only one object that we need to update which informs the system of the update.

Thus there only needs to be one method which takes as parameters the composition, the Tal and a way in which to specify the size of the composition. The main way that the first two parameters are specified is much like the way in which the Tal has been specified - use of an enum. By having such a property, we can force the developer to explicitly declare the Tal they want which is recognised by the system appropriately rather than relying on user input. This is also true of the composition and thus to keep modularity and consistency the enum has been implemented in the Composition Generator.

Each composition has been designed to generate only one version of the composition (apart from the Kaida). The Kaida was different as it incorporated many different sections and thus the variable part was the number of variations. Therefore it was considered better to parameterise the number of variations the Kaida class generated.

Therefore the composition generator uses the specified value passed as the number of times to generate a particular composition. In the case of the Kaida however, the number of phrases indicates the number of variations it generates. On each generation of a composition it is appended as a child of the root node of the first tree that was generated thereby create a large tree of multiple compositions.

The player object therefore has the role of using the composition generator to specify the compositions it wants to play and of what size. Again, re-iterating the notion of no artistic influence used within this system, the player object will be implemented to use a pragmatic method of choosing compositions to generate an entire performance.

Since we are using user input to tailor the performance to some extent, we shall define the

Table 4.1: Table for a Solo Performance

| **Section/ Mood** | **Heavy** | **Medium** | **Light** |
|---|---|---|---|
| **Start** | Prakar 3 | Prakar 2 | Prakar 1 |
| **Middle and End** | Kaida 7 | Kaida 5 | Kaida 3 |

Table 4.2: Table for a Accompanist Performance

| **Section/ Mood** | **Heavy** | **Medium** | **Light** |
|---|---|---|---|
| **Start** | Prakar 3 | Prakar 2 | Prakar 1 |
| **Middle** | Prakar 7 | Prakar 5 | Prakar 3 |
| **End** | Mukhada 1 | Mukhada 1 | Mukhada 1 |

performance parameters as specified by the requirements and show the design of how the player object will use them to generate the performance. The performance parameters are:

- Composition Type: Solo or Accompanist
- Tal: Teental, Jhaptal, Rupak, Dadra
- Mood: Starting mood of the performance

To translate these parameters into a performance we must first construct a structure for the performance in which the parameters can effect. As a result we can create the following form: beginning, middle and end. This can enforce the correct use of the composition, as they have been grouped as cadential and cyclical compositions. Therefore the beginning and the middle must use cyclical compositions (Theka, Kaida, Prakar) and the end section must use a cadential compostion (Mukhada, Tihai).

To keep the simplicity of the performance, a simple lookup table can be created to generate the selection of the compositions. The auxiliary element that has been kept in mind for this table is if the mood is heavier, the longer the compositions will be.

In tables 4.1 and 4.2, the elements that are selected are the composition and the number of phrases the composition should have. The solo comprises mainly of the Kaida as part of the solo as the variations are the improvised, and variant structure. In addition, since the Kaida has an ending, the solo table does not require the ending to be selected.

The lookup table can return a struct which can contain this information and can be easily destroyed after its use. Thus the struct: PerformanceMetaData was created to hold this information and is destroyed after each section of the performance has been created.

Although this is a rigid way of implementing the selection of compositions are selected, it seems the most intuitive way in which to choose compositions based upon the understanding of their role as an accompanist to another instrument or solo tabla. When this was first implemented it was found that there was some similarity in the performances generated and thus a better sense of interaction between the three sections was needed. Therefore the mood of the player would change throughout the piece dependent on the previous composition and it's size. If the previous composition was large, then the mood can decrease to release tension. Likewise, if the composition was small then the mood can become heavier and increase the potential size of compositions to create tension.

This therefore created better dynamics within the performance and a sense of realism, without moving too far away from the theoretical compositions.

The player object appends the trees generated by the three sections of the performance together and has a public property which points to the root node of the first composition thereby having only one reference to the entire performance.

On generation of a new performance, the instanited player object is destroyed before a new one is created to stop any memory leakage.

### 4.3.7 The Interface

The interface was the simplest part to design due to the trivial requirements of getting the user's input and displaying the completed performance. The initial design of the layout of the interface can be seen in Figure 4.10.

As we have designed the system to be MVC, the interface only needs to instantiate one object: the player object and pass the performance parameters to it in order for the composition to be implemented.

Once the player object returns the performance tree, a simple tree traversal algorithm can identify the beat nodes (i.e. nodes of type 2) and print the bols for that beat (the child nodes for a beat node) indicating the bols per beat.

For the audio playback, a simple algorithm has been created which will enable the correct output

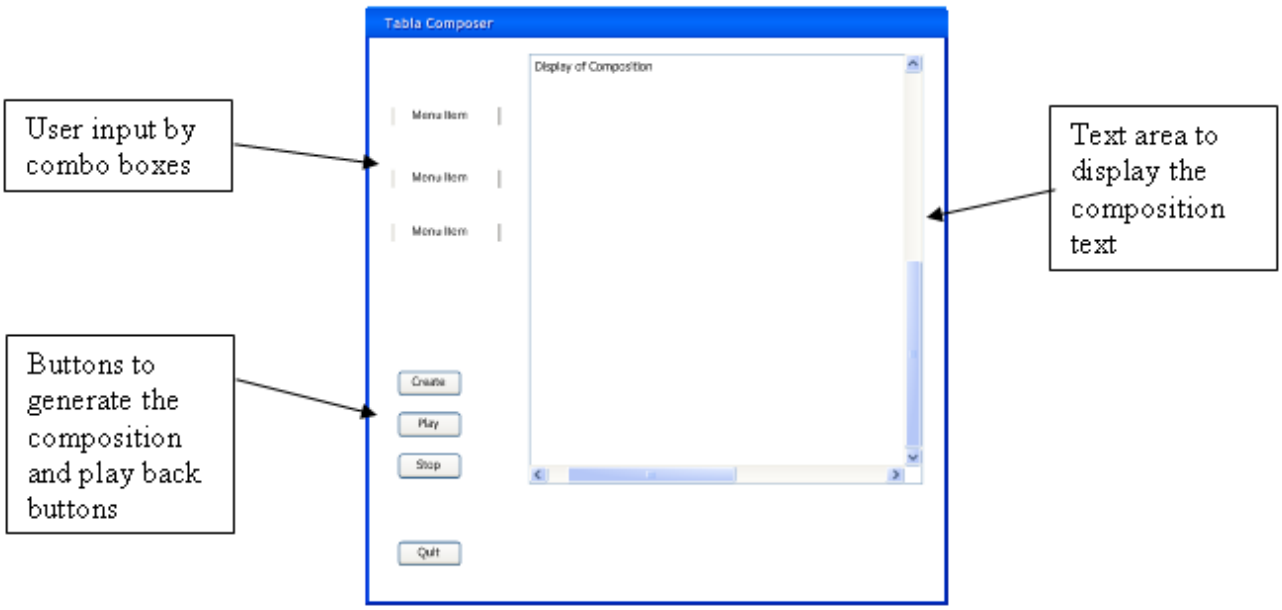



Figure 4.10: Design of the interface

of the bols assuming there are samples for the bols. A basic way in which tempo can be set is to use the BPM system. This ties in well with the way in which we have designed the composition as all beats are attached to a node which represents a single beat.

Therefore if we know the BPM, then we can calculate the duration of a single beat and thus divide this length of time equally amongst all of the bols on that beat. Therefore an algorithm can be implemented which can traverse the tree and by identifying the beats can calculate the time required for each of the bols. A loop can then be created which iterates through each of these calculated times which corresponds to a bol and playback a WAV sample for that bol and sleep until the next bol must be played. The term sleep has been used as the intended implication for this loop is that it runs as a separate thread to the rest of the application so the interface is still usable while the composition is being played.

The main problem with this implementation is finding an appropriate API from the C++ language which also allows a multi platform capability. Unfortunately no such API was found and due to time constraints a windows oriented API was used: mmsystem, which allows the playback of WAV files asynchronously. It was thought that it would be better having this feature

partially functional to show the applicability of the compositions is better than not having the functionality at all.

In addition to this, the literature review explained that at the use of certain bols can sound different due to the context it is in. The example that was given was the bol Ti would sound different to when it is used in TiTe. However as we have been dealing with the grammatical modeling of the compositions using the linguistic connotation of the language, this section will need to be revised further to create a better audio interpretation of the performance generated. This current method has been adopted primarily from the TaalWizard application [15].

This now completes the discussion of the design and implementation of the system. By having a divided section between the design and the implementation, discussing the iterations and the adjustments to the original design would have been difficult to expand upon and illustrate.

# Chapter 5

# System Testing

The testing for this system is not quite a formal method. This is because testing the system has been constant throughout the development. This is mainly due to the fact that Tabla compositions were still being understood through design and development causing constant checking and modifications to the compositions. In addition, due to the modular design of the system, after each module was implemented, it was tested against the rest of the system. This therefore has allowed a better way in which to develop the system and make changes from testing, which were discussed in the design and implementation section.

In this section we will want to discuss the extent to which the system adheres to the requirements laid out in the requirements specification. It would be advised that the analysis and the specification be re read to refresh the reasons behind the development of the specification from the issues and ideas raised in the literature review.

In order to properly validate the system and test it we shall use the structure of the requirements to see how well they have been met. To ensure that the compositions are validated and that they are correct it will be compared to existing compositions and they will also be verified by professional Tabla players. Output of the performance can be found in the appendix. The test plan will also take the form of the requirements to keep simplicity and consistency.

## 5.1 Testing of Functional Requirements

### 5.1.1 Grammar Modeling and Transformations

1. The system is capable of creating phrase structures which are direct mappings of the Tal and can be easily inserted into the system. This can be done by an analysis of the Theka composition from which production rules are created and they can be appended as rules to a phrase object which holds all rules per Theka and uses a recursive descent method of adjoining the rules and creates the tree. The development of the tree has been also tailored to be an N-ary tree and has also the simple capability of adding and deleting child nodes.

2. The Theka tree that is generated is a lexicalized phrase structure tree. The system has been designed so that the Theka can be generated without the bols and therefore represents the theoretical idea of the Tal. Each of the compositions either use the Theka tree to perform the transformations on or uses the un-lexicalized tree to append bols to. If the composition appends bols to the tree a set of bols can be entered a single bol at a time and in a particular order - or as another phrase structure grammar to generate a tree to hold the bols which is then used to perform transformations on and then always append to the Tal tree. The tree has also enabled the ability to preserve the theoretical structure of the Tal by the use of attaching types to the nodes of the tree.

3. The transformational object is used to perform the transformations using a single function which takes the structural change and structural index as parameters as well as the root node from which it works from. The transformational function works only on the children of the root node provided since there is a fixed structure of the tree having a set of levels (i.e. sentences and beats) and thus the need for a pattern matching technique for the structural index is not required as such. Therefore the way in which the structural index works is by using numbers to identify the child nodes of the root and then perform the structural change upon. Each of the compositions implemented that uses the transformation object ensures that it uses a basic structural index to identify a subset of the child nodes and uses some sort of traversal algorithm to move about the tree and use the transformational function as specified. In other compostions, the bols have been appended to a tree which is generic for all Tals and thus the transformation object can be used upon the tree. Therefore the transformation object is a generic function which can be used upon the child nodes of a given root and perform the four basic functions of transformations: permutation, substitution, nulling and insertion.

Each of the compositions uses a lexicalised phrase structure tree to generate the composition. Although the system successfully does this, it does not always use the transformation object to do this. This is because although a transformational rule could be made for the specific transformation required, some initial analysis of the phrase structure had to be performed before the structural change could be created. This is true in both the Kaida and the Tihai. The Tihai makes little use of the transformational object but uses a transformation in upon a tree which yields the desired composition. The definition of the transformation that is implemented by the Tihai composition will be in the next section. Therefore this requirement has been satisfied, but also the results have shown an extension to this.

4. The compositions that have been generated are verified to be correct. Each of the compositions implemented are used for their required purpose and adhere to the rules defined by the composition. There are inconsistencies which are generated in some parts of the Kaida described by David Courtney. This inconsistencies are breaches of the rules dictated by the Kaida for reasons which will be explored in the evaluation section, as well as an in-depth analysis of the other compositions.

5. Modularity of the system has been successfully implemented. Each composition has been implemented as separate classes and thus can be extended easily by implementing a class which firstly inherits from the Composition Object class. Secondly, the composition generator is made known of the existence of the composition and thus can be used to instantiate the newly implemented composition.

   In addition to this, the transformation object has been made in a separate class and it's use is not obligatory by the compositions. Therefore each of the compositions are free to utilise other techniques and functions to perform transformations or any other tree manipulations. The modularity of the compositions has filtered throughout the entire system making use of the Composition Generator which consolidates all of the compositions used by the system. Also by using the player object to consolidate the actual performance and to interpret the performance parameters into tailoring the compositions it creates a final tree of compositions conforming to a performance.

### 5.1.2 Global Structure

5. The performance parameters have been defined as follows:

- Composition Type: Solo or Accompanist
- Tal
- Starting Mood of the Performance

This has translated into a way in which to tailor the compositions which is performed by the player object. The object uses the idea of having three sections to the performance and selects appropriate compositions using the performance parameters against a basic lookup table to supply the parameters to the compositions objects. Therefore the player object from a higher level uses the Composition Generator as an engine to return compositions on request, making the global structure of the entire performance easier to control.

6. The lookup table provides the appropriate information for selecting the compositions dependant on the section of the performance (i.e. beginning, middle or end). If the system was to be extended then this lookup table would need to be extended as well. However it fulfils the requirement to select the compositions appropriately. Thus an accompanist performance uses predominantly cyclical compositions and ones which reflect the Theka and the main elements of the Tal. Therefore the Prakar and Mukhada are used as cadential pieces to end the performance. The solo performances also use a cyclical composition but a different type and must also use ways in which to generate tension and improvisation such as the Kaida. Therefore the compositions are used for their intended purpose within a performance.

### 5.1.3 Graphical User Interface

7. The interface gives the ability to select the three different performance parameters including Tal and type of performance. The third parameter was created for the piece to have a starting point reflecting mood. This is carried out by the use of combo boxes thus forcing the user to select from a given choice as there are fixed names for each of the options as they are referenced by enums. This also enforces correct selection of the options and does not give rise to error on user input.

8. The interface uses a button which generates the composition based upon the performance parameters selected. If the user does not choose any of the performance parameters then the default options are used which generates a performance. This is again so to reduce an error message appearing. The performance is displayed on a text editor control which is non - editable. The tree traversal algorithm walks over the tree - depth first, to prints the bols to the text editor. A delimiter character: '|' is used to group the bols which are played

upon the same beat.

On click of the button again, the text is cleared off of the text editor and the new generated performance is displayed based upon the present performance parameters. In addition there is a play button and a number control which allows the BPM to be set. Therefore when the play button is clicked, the composition is played at the BPM set. This defaults to 120. The user also has the ability to stop the playback.

## 5.2 Testing of Non-Functional Requirements

### 5.2.1 Performance, Extendability and Error Handling

9. When an object is deleted, the memory is freed appropriately. When the user generates a new performance the player object is destroyed and by the memory management and use of destructors which will free space which has been allocated and will destroy objects.

   Therefore each of the compositions that are created for the performance are destroyed before the new performance is generated. There is a slight memory leak which has extended from the Qt API. This is mainly to do with the text editor which is used to display the text and the object which holds the text. The leak is minimal and has been reduced though does exist.

10. The entire system has been designed to be modular. Therefore the use of the composition object has allowed the composition to share common methods and properties one of which is the phrase structure grammar. The phrase object allows the easy implementation of the rules and thus the creation of a phrase structure tree. This has inherently become the way in which the Theka is automatically created or if unlexicalised, is the theoretical structure of the Tal. Insertion of another Tal can be made simply creating a new phrase object and appending the phrase structure rules of the Tal to it allowing easy extendability of Tals.

    As described previously, by designing the system to have each composition in a separate class enables the modularity of the system. Extending the number of classes for each composition allows the easy expansion of the performance and in essence, the grammar. Separate classes also imply the variances of the transformational methodology used in each composition thus giving each composition freedom to explore different techniques to create the composition and not binding it too tightly to one method. By extending the

compositions the composition generator is also extended. Therefore by having the player object and the composition generator as separate object maintains the modularity of the entire system and not just the Tals and compositions.

11. All of the errors are handled internally. No errors due to generation appear. At points when the system cannot generate a composition or finish a computation due to reliance of a random number generator for example, a heuristic approach has been adopted which enables the quick rectification of errors and execution of procedures correctly. An error message is returned to the user if they choose to play a composition before one has been generated. The error message informs the user that they must generate a performance first before choosing to play it.

## 5.3 Further testing

This section refers to the full requirements specification in the appendix. It is recommended that the reader revises this section in order to understand the requirement being tested.

### 5.3.1 The audio output of the performance

Requirement 1.5.2.3 states the implementation of a playback system which will use samples which will be assigned to each of the bols in a performance and be played out to the audio hardware of the computer. The mmsystem API which provides the appropriate libraries to allow this feature to be implemented. The user has the ability to stop the playback which works successfully.

### 5.3.2 Usability

This is satisfying requirement 1.9. The system has been made usable by the implementation of the graphical user interface. There is a simple way for the user to select the performance parameters and generate the composition and to hear a playback of it. The use of specific controls on the interface allows the easy usability of the system. In particular the use of the combo boxes to minimalise the error introduced by user input, the input controls in one area of the interface.

### 5.3.3 Cross Platform

The system has not been fully tested on multiple platforms and therefore this requirement has not been fulfilled. However, the system has been developed in C++ which is highly portable across different platforms. The only feature of the system which may cause disturbances in the cross platform functionality will be the playback functionality, though as this is a incomplete feature can be re-factored. The re-factoring of the code in order to make it fully cross-platform should be extremely minimal.

### 5.3.4 Extendability

The system should be easily extendable due to the nature of the design of the system which has been described previously.

# Chapter 6

# Critical Evaluation

This section seeks to evaluate the system that has been created. It will take upon the same structure as the design section and give a description as to whether the method chosen was the best solution for the particular problem in hindsight.

## 6.1 The Phrase Structure Grammar and the Transformation Object

The phrase structure grammar constituted the main building block of every composition implemented as intended. The most useful feature that was found in the design stage was the way in which the grammatical modeling of the Theka gave way to the construction of the beat and sentence structure. This is not a complete surprise since we wanted to model the Tal for compositions to exist in and this is the main purpose of the Theka composition. However, the Theka has the extended feature of bols which define the clap/wave structure.

The tree structure that the phrase structure gives allows the simple and most natural modeling of the Tal. As a result implementing the ability to generate the Theka unlexicalised proved to be an extremely useful tool. For example, a Kaida Theme has no specification as to how it connects to a Theka/ Tal and is left up to the interpretation of the player. As explained in the design section, the system calculates the connection to the Tal by finding the layakari the theme can be played in one cycle of the Tal. Although this is theoretically valid I shall later show the problems with this technique in my discussion of the compositions.

The use of the single phrase object to hold the phrase structure grammar has allowed flexibility and also the simple ability to extend the number of Tals. It was also found that Kaida themes could be created in the same way showing its use in various compositions. Using the recursive descent method seemed the most natural way in which to build the tree from the rules. The method utilises two functions to complete this task. Although this method works, it could be re-factored in order to make slightly more elegant. However, as it was not a priority and despite the problem highlighted in the design section, of the extra node having to be deleted when the tree is constructed, it has remained the same and has not caused any problems.

The transformation object has been the most volatile and perplexing object to create. The reason for this is due to the fact that it was uncertain whether the transformations used for linguistics was actually complete for the transformations required for the language of Tabla compositions. The fundamental difference between the transformations laid out by Chomsky and the transformations that were used for this language is the requirement to dynamically create and/or apply the transformational rules. This is due to the fact that unlike linguistics, the tree structure only has a specific semantic of sentences and beats and there are no semantics attached to the bols. Therefore the use of a structural index by pattern matching nodes (identified by their type or value e.g. noun phrases, verb phrases etc) were not possible in this language. Therefore, the transformational object was confined to identify child nodes from a given root node and to perform the basic functions upon those nodes.

This has proved to be a possible way in which to create the transformations as compositions have been created using the object successfully. However after the development of the compositions, this object may be altered to perform more powerful transformations which could allow better ways in which to develop the grammar and the transformational rules. It would have been better to implement the transformation algorithm after gaining a better understanding of Tabla compositions and performance. Despite there was a formidable understanding of the compositions throughout the design, the development of the compositions highlighted particular areas which could have been made easier. For example, having a specific way in which to substitute a bol for its Khula/Band homomorphism counterpart would be a useful tool, the reasons for which will be described later in the discussion of the Kaida. It was also found, although this did not necessarily cause a problem, that there could be some interpretations of the structural index which would not always work, such as performing a substitution and a inserting simultaneously. Therefore the structural index could have been implemented by using a small language whereby the string could have been parsed and formalised by a grammar. This would make the transformation object extremely powerful and easily extendable. I would also allow the ability to

build in the other transformations that were discovered throughout the implementation such as the ChangeLayakari() algorithm or the AppendThemeOfBols() algorithm. These were functions which had to be developed on top of the basic transformation algorithm as they manipulated the tree is a different way in which the transformation function does.

This is not to say the current implementation is incorrect or is not sufficient to develop transformational rules for compositions. The implementation of the four basic transformational methods have been satisfactorily designed and have been able to perform the required task. However, building a more flexible transformation object would allow a more powerful tool to use and therefore would allow a more consistent implementation of each of the compositions. The reason for increasing the power of the transformation funtion may be important is due to the fact that there may be transformations required for other compositions which are unknown. As a result the creation of transformational rules may be more complex and would require a powerful transformation object greater than the one currently implemented.

The use of such a method as AppendThemeOfBols highlights the mixture between the Tree Adjoining grammar and the Transformational Grammar as discussed in the literature review. Here we have a sequence of bols that have been grouped together and exist as a tree. However after some further analysis, it was found that the bols did not have to exist as a tree, but that the ordering of the bols only had to be maintained as the sequence did not specify the relation to a Tal or layakari. The tree structure was generated as it became an easy way in which to create these sequences. An example of the was the Bag of Tricks [9]. This has introduced the notion of a knowledge base within the system and that parts of the language is constructed from preset 'words' or valid sequences. The creation of these sequences may not matter a great deal as the a Tabla player can be taught these sequences as they are, without any reference as to how they were originally constructed or why.

Although the current implementation of each of the basic transformation functions are satisfactory, the largest problem was the insertion function due to the fact that it complicated the tree structure and essentially altered the structural index. However the solution implemented should be complete as it is though this will still be an existing problem even if the structural index 'language' was implemented.

## 6.2 Analysis of the compositions

As the author is not a Tabla player, an analysis must be made of the compositions generated in order to verify them. Output of the system can be found in the appendix along with an example of the grammar that the system either implemented or used to generate the performance.

The compositions that have been created were tested for validation after they were created one at a time to ensure they were producing the correct sequence and also that the generation of the compositions were responding to the performance parameters. As we have already discussed the global structure of the system is controlled by use of the player object. We will not be discussing that here but instead we shall be mainly concentrating on the compositions individually. In addition to my own observations, David Courtney and Jim Kippen have kindly given their analysis of the compositions which I will be making reference to.

### 6.2.1 Theka

As this composition has been used as a way in which to generate the Tal, its generation exists for every Tal implemented. This will always have a fixed structure and therefore will never change.

### 6.2.2 Prakar

The idea of the Prakar is that it puts more 'flesh' on to the Theka. It makes the fairly trivial idea of the Theka more interesting and less arduous to play for the Tabla player. Therefore as it can be argued that the Prakar is a derivation of the Theka, the use of appropriate transformations should yield a Prakar composition.

The system has been successfully able to use the Theka tree and apply transformational rules to it in order to yield a Prakar composition or what can be identified as a Prakar composition based upon the definition of the Prakar. The Prakar is the most basic example of using the transformational rule to create the composition as it instantly derives it from the Theka. Courtney has verified that the strategies used to generate the Prakar are valid and that the Prakar composition is correct. The strategies used within the performances are in conjunction with the linear stochastic process whereby the further use of strategies moves more towards a greater use of techniques which increase tension. For example, the first performance in the appendix uses the basic strategy of doubling the final beat of each sentence. However as the piece progresses, it begins to use more obscure techniques (such as using the Bag of Tricks) which is a good way in

which variations and improvisations work. Courtney does mention however that in some cases in the use of the bols from the Bag of Tricks, the use of the pause 'S' is redundant as it makes no contribution to the composition, i.e. it's absence would make no difference to the composition. For example in the first performance in the appendix we have a particular Prakar:

$| Dha \mid Dhi \mid Dhi \mid Dha$
$| Dha \mid Dhi \mid Dhi \mid Dha$
$| Dha \mid Ti \mid Ti \mid Ta$
$| Ta \mid Dha\,\mathbf{S} \mid TiRa \mid KiTa$

whereby the 'S' which stands for pause does not need to be there. This does not necessarily change the actually nature of the composition, but it's presence has no meaning and this is overlooked by the system.

### 6.2.3 Kaida (and Tihai)

Since the rules of the Kaida are so strict, verification as to whether the correct piece has been generated has been fairly simple to check although there are issues to point out.

The first element that must be checked within the Kaida are the variations since these are dynamically created. The only variation which has been 'hard-coded' is the Dora which is the first variation. Thus on analysis of the other variations, the rules of the Kaida are generally upheld. The rules being; the use of the bols of the theme, correct permutations of the bols, the variations showing a Bhari/Khali structure. In addition there must be a final Tihai composition which is based upon the bols of the Kaida Theme.

Each of the variations are different on each creation of the Kaida as the variations are based upon dynamic transformational rules, created randomly at run time. Thus this introduces better variances and uniqueness in each of the performances generated. Although the other compositions utilise the phrase structure tree, the transformational rules actually apply to the structure of the Bhari/Khali phrase structure tree. This has actually revealed an interesting aspect of the language. It has shown that there are possible multiple phrase structure grammars that could exist within the language in addition to the Tal which was origianlly used as the phrase structure. This also highlights the vast contrasts between compositions. This may be due to the fact that Tabla compositions have been developed and have evolved from many different areas and

Gharanas (stylistic schools) and as a result various methods have been adopted. However, the phrase structure for this composition is fixed for all Kaidas unlike the Tal phrase structure.

The first sample performance usees the Kaida in Teental (16 beats) The second sample performance is in Jhaptal and also uses a Kaida. From the comments made by Courtney and Kippen, there are some issues that have arised from the compositions.

The first is made by Courtney whereby the Bhari/Khali structure is not shown in the variation:

$| DhaDha \mid TuNa \mid DhaDha \mid TiTa$
$| DhaDha \mid TuNa \mid DhaDha \mid TuNa$
$| DhaDha \mid DhinNa \mid DhaDha \mid TuNa$
$| DhaDha \mid DhinNa \mid DhaDha \mid DhinNa$

This does not work as the Khali structure is not properly reflected despite the third and fourth sentences of this variation are part of the Khali section of the theme. The problem lies in the transformational rule and choosing the correct Bhari section of the Kaida tree. This is because the Bhari structural change string for this is: "2,1,2,2" using the example used in the design and implementation section. Although this is a valid variation, the corresponding Khali structural change string is: "6,5,6,6" which again is valid. In Courtney's terminology this would be BABB-BABB which would be allowed in general terms. However due to the nature of the Kaida theme this has yield the variation above where the corresponding Khali section does not use any of the Band bols thereby generating an incorrect variation.

This is a rare occurrence for the system to make but such a variation can occur. The variation could be made better if another transformation was performed on the third sentence which would replace the bols with the Band bols (indicated in the Khlua/Band homomorphism) [18]. Thus this would produce this varition which would be valid:

$| DhaDha \mid TuNa \mid DhaDha \mid TiTa$
$| DhaDha \mid TuNa \mid DhaDha \mid TuNa$
$|$ ***TaTa*** $|$ ***TiNa*** $|$ ***TaTa*** $|$ ***TuNa***
$| DhaDha \mid DhinNa \mid DhaDha \mid DhinNa$

This shows that the transformational rules implemented for this Kaida composition is not com-

plete and that an extra transformation(s) must take place in order to verify that the correct bols are used, something which was not understood until completion of the implementation section.

The second issue with in particular with the Kaida again is the use of bols when choosing sub-sentences for the second level of variations identified by Kippen. After the sub sentences are generated the combination of some bols are either rarely played or may not actually work such as 'Ti Ti Ta' OR 'Ta Dha' at the beginning of a sentence. This now comes to the part of the compositions which was not possible to gain a good understanding of since dexterity issues were not explored in the theory of compositions and therefore these are the next issues that the system needs to deal with if the project was to continue development.

This idea is again re-iterated in the generation of the Tihai. The Tihai seems to be correctly generated using the Kaida Theme and correct selection of the bols for each Palla. The Bharan is also used sensibly as sometimes it may not use the entire of the Tal and share with the Tihai and if not possible, will use a whole Tal for the Bharan and the Tihai separately which is allowed in the Kaida.

Courtney explains that the Tihais generated are always theoretically correct as intended, though sometimes the bols that have been selected can be difficult to play practically and hand movements may be difficult. This introduces the aspect of practicality and theory in the generation and playing of compositions. Although we can generate compositions which work in theory, they may not be practical to play terms of dexterity. The Tihai composition is considered the composition implemented weakest. It makes no use of the regular transformations and little use of the fixed transformations. This is due to the fact that the transformation object does not specify over multiple layers of a tree for the structural index. The generation of the Tihai utilises an entire tree to select units for the palla which over extends the reachability of the transformation function. Time did not permit this extension to the transformation object. It is still considered a transformation as it uses the Kaida's phrase structure tree passed to the Tihai class and though does not manipulate the tree, uses this tree to build a new tree. However, if the transformation object was extended, a specific transformation rule could be used to perform the same task.

Courtney later explained however a more general way and simpler way in which to generate a Tihai using a Kaida's phrase structure. Using his example of the A and B structure, one could use them to make a Tihai such as: AB pause AB pause AB pause or B pause B pause B pause. However it would need to verify that these structures would enumerate the correct number of bols required.

The idea of theoretical verses practical compositions is highlighted further in the generation of the Rupak Kaida. The third sample performance in the appendix is in Rupaktal. This particular example has been chosen as it shows that the entire performance is theoretically correct but is far too impractical to play. As the Kaida Theme does not specify the connection to the Tal, the system has created this particularly large Kaida to be in one cycle of a Tal which is not always necessary. This is mainly due to the specific phrase structure that builds the Bhari/Khali tree. In this instance, the Bhari/Khali tree must be expanded greater than one cycle of a Tal in order to create the correct tree specific to the Kaida. This may be a simple method of realising that if the number of bols within the Bhari/Khali section is too great, the number of bols per section must be reduced and the tree must be expanded. Therefore if we have 8 bols per branch of the Bhari/Khali tree, we can consider this too many and thus restrict no more than 4 bols ber branch and have 8 branches denoting 8 sentences to the phrase structure tree. This will give a better tree to permutate realistically. The current system would require minor changes for this to be possible. In particular the creation of the tree (AppendBolsOfTheme() method and the PerformPalta() method) which sets up the command DoVariation() and passes the structural index and structural change strings.

Other than these issues, the system has generated the Kaida and the Tihai compositions correctly and honestly.

### 6.2.4 Mukhada

This follows on from the discussion of the Prakar composition as similar strategies are used. By performing the transformations on a tree it has been able to successfully use the Mukhada rules and generates a build up of tension by using bols from the Bag of Tricks or a Kaida theme. If this composition is extended it can use more strategies to build up tension.

The most useful part of this system has been the modularity of the architecture. Since the language can take many variations, extending the grammar implies extending the number of compositions which is explicit here.

As further transformations were discovered throughout the development process, the transformations could be classed into two main groups:

- Regular Transformations: As described by Noam Chomsky whereby we have a structural

index and a structural change specifying the operations to manipulate the tree - using the four basic methods (substitution, permutation, nulling and insertion).

- Fixed Transformations: These transformations which may or may not use a specific combination of the basic methods to manipulate the tree - altering arrangement or adjoining other nodes to the tree in a specific way. These are generally fixed transformations and do not vary dependent of the phrase structure.

The fixed transformations (including AppendThemeOfBols(),AppendBolsToTree(), ChangeLayakari) perform general transformations which could be encapsulated by the transformation object, but since these are essentially large scale it may be difficult to use the transformation object. These transformations traverse the tree in a particular way different to how the regular transformation would work. For example, the AppendBolsOfTree() function works backwards through the phrase structure tree and appends bols from a tree in the Bag of Tricks which is quite a unique and individual transformation. Replicating this using the regular transformation method would have been a difficult transformational rule to create using the structural index and structural change.

The method of analysis of the compositions has been fairly informal though due to the conditions of this project it has been enough to analyse the compositions. A better way in which the analysis and development of this system could be executed is by using direct help of an experienced Tabla player who understands the principals behind each of the compositions well. A closer relationship with such a person can allow a quicker and better understanding of the compositions which can increase the speed and precision of the design and development.

## 6.3 The Composition Generator and the Player Object

The composition generator has served a great purpose as an engine to return compositions and as a way in which to consolidate the compositions. Although this was it's intention from the start, its appropriateness was not realised until used in the player object. The two objects have worked hand in hand and have been a useful way in which to control the global structure of the performance. Using an object oriented language and design has made the implementation of these much easier.

As explained in the requirements and in the design section, the player object merely consolidates the entire performance and is a fairly crude implementation. Despite this, it performs the

basic requirement of controlling the global structure and forcing the variations of each performance to be unique without breaching the bounds of theoretical compositions apart from the ones stated about the individual compositions.

## 6.4 The Interface

Using the MVC method of implementing the interface has allowed a front end to be developed rapidly which performs a basic task as specified by the requirements section and can be easily extended.

Although fulfilling the requirement of making the system a multiplatform application, there are parts of the system which may cause problems, in particular the playback functionality. However, as for reasons stipulated in the design section, it was implemented as a partial complete feature which can be enhanced and does not restrict the main focus of the project.

A large amount of time was also dedicated to the necessity of learning the API for the interface. Despite its simplicity and the various tools that it offers as a development package, a simpler or even a familiar GUI API may have been quicker to implement. However, if another API was chosen, it might have sacrificed the ease of transferability between platforms.

# Chapter 7

# Conclusion

*"A good drummer listens as much as he plays."*

**-Indian Proverb**

The main intention of this project is to develop a system which uses a derivation of an existing formal grammar(s) and to use it to model the language of Tabla compositions. The system was to generate a set of compositions based upon some preliminary parameters and to be able to generate the same compositions in any Tal.

In order to make this a usable system, a front end was to be implemented whereby the user could select the parameters and the composition that is generated would be displayed (as text or audio samples) back to the user. This would serve the purpose of being an intelligent system as well as serving a practical purpose for the listener.

As shown from the system testing and the critical evaluation sections, the system that has been developed meets most of the requirements and has been able to deliver a system which uses a derivation of the transformational and tree adjoining grammar to develop the language. However although the literature review and design highlighted these were the best techniques to use for modeling this language the techniques have raised some issues.

The literature review found that the phrase structure grammar would normally be used to model a natural language unless there was an issue of cross dependencies which Chomsky solved by using the transformational grammar [7]. The phrase structure tree is lexicalised by the semantics

of words and then transformations can be applied to generate an appropriate sentence. For Tabla compositions a lexicalised phrase structure is constructed to hold the main cycle of sentences, beats and bols. Transformations are then applied to this tree which construct the desired composition. This shows that the intended grammar for the language of Tabla compositions uses the same idea as used for linguistics.

It was found that in addition to the transformations required to create the compositions, other more specific transformations were discovered. The critical evaluation highlighted the vast differences in the use of transformations for each of the compositions. This may be attributed to the fact that Tabla compositions have evolved from numerous geographical areas and under many different Gharana's (stylistic schools) which use different methodologies of teaching and style as discussed in the literature review.

The critical evaluation also highlighted some areas of the compositions generated which were illegitimate (specifically the Kaida) but this mainly was due to the lack of understanding of the Kaida rather than incorrect generation. This implies that further understanding is required despite the work around solutions that were suggested in the previous section which would be valid.

Despite this, it seems that the transformational grammar, phrase structure grammar and an aspect of the tree-adjoining grammar have been able to model the compositions appropriately along with the development of some specific transformations. Although we have modeled this 'linguistic', the language is musically connotated rather than semantically which may be on of the reasons why other more unique transformation functions were implemented. However the connection between the language of Tabla compositions and linguistics is closely related and we can see the comparisons by comparing the ways in which the both languages use transformational rules.

It has also shown that to extend the grammar for this language we would have to extend the number of compositions. This shows that the language is a set of smaller languages all of which are interconnected by the use of Tals. Although students learning to play the Tabla may have not be taught this, the modularity of the language allows for further extensions to be made to the grammar to increase the size of the language.

Using transformations implies there is some formal structure created which is manipulated. The grammar developed here shows that each compositions describes different transformational rules though all fundamentally work with the same semantics showing the formalism of the language.

It is the formalism of the language which this project has been intended to show which it has done. This is not to say the grammar models the cognitive thought of the player in artificial intelligence terms, but models the underlying development of a composition.

In addition the transformation rules that are developed are not just for theoretical purposes but also give the ability to create artistic influences. This allows the explicit separation of theoretical and artistic compositions which is a powerful tool.

The testing and critical evaluation sections have also shown that another set of transformations may have to be implemented to verify that the bols generated allow practical playing. Therefore, the transformations in this language are used for three purposes:

- Create a theoretically correct composition
- Create artistic influences over the composition
- Ensure practical playing

They do not necessarily have to be executed in this order and they may even overlap as one rule, but these are the main high level rules that should be considered which this project has shown.

The project has served the purpose of showing the initial steps towards building a more advanced system and highlighted the issues and problems which can occur. It has shown that the language of Tabla compositons can be formalised as a grammar showing the strict formalism that exists in Indian classical music.

It has also shown the complexity that exists within Indian classical rhythm and it's connection to linguistics which has been a fascinating journey. I am encouraged by David Courtney's and Jim Kippen's analysis of the compositions and of the system as a whole. Their feedback has promised that despite small modifications which are required, the grammar is a good starting point to develop from and that progress has been made.

## 7.1 Future Work

There are numerous works which could take place beyond this project. There are firstly the changes and improvements that can be made to the transformation object as previously mentioned. The system can also be made more robust by developing further compositions based

upon the basic phrase structure grammar and developing the transformations rules for each of those compositions. The system can be used to compare the differences between compositions for different Gharanas or Tabla players by comparing transformational rules. This also implies creating transformational rules and extending compositions to encompass artistic influences.

Further analysis of a Tabla players performance can be analysed to observe the variance in their performance. This analysis can be made to improve the player object implemented in this system. This could be considered as an artificial intelligence analysis and from a computer music perspective. This also allows the possibility of truly attempting to model the cognitive thought process of a player in choosing compositions and developing them.

Apart from improvements and refining the system, we can begin to develop the system further by using other methods of generating the performance such as a real time accompanist to a melodic instrument. This would generate a performance which would match the music produced by a melodic instrument and would change the nature of the performance dependent on the way in which the melody altered. Various modifications upon these ideas can also be considered.

Kippen has commented that the interface may require work and that the samples used need to be developed in relation to the composition generated. Having one sample per bol is not enough for an entire performance and therefore work would need to be require to match the use of different sounding bols within a performance.

## 7.2 Final Remark

This project has shown me the vast complexity and intricacy of Indian classical music and to appreciate the work and dedication Tabla players give to learning and mastering the instrument. It has influenced my understanding and thinking about all types of music as a musician and computer scientist.

I feel that the project can serve a great purpose to understanding the abstract theory of Indian classical rhythm and as a practical tool for those who wish to learn how to play the Tabla as well as using it as a tool to play with or beside. I think more can be learnt from this system if developed further and can contribute a large amount of interesting ideas to algorithmic composition

specifically in Indian classical music since there is a lack of such research.

# Chapter 8

# Special Acknowledgments

I would like to give special thanks to firstly David Courtney Ph.D. author of [9], [10] and [11] without whose advice and expertise, the completion of this project would not have been possible.

I would also like to thank Jim Kippen Professor of Ethnomusicology at the Faculty of Music, University of Toronto, author of [18] who worked with Bernard Bel to develop machine learning systems as tools for research into improvisatory strategies used by tabla players, whose criticism on the application has also aided the completion of this project.

Both have written exceptional literature on Tabla compositions have provided a foundation for understanding Indian Classical Music for western listeners.

# Chapter 9

# Glossary of Indian Classical Music

**Alap** A slow rhythmless elaboration upon the rag used by vocalists and instrumentalists.

**Avartan** A cycle of a Tal

**Bhari** The section of a Kaida which uses Khula bols.

**Band** Non-resonant bols

**Bols** The mnemonic syllabi of the Tabla

**Dholak** A crude folk drum characterised by a cylindrical wooden shell covered with skin on both sides

**Dradtal** A Tal of 6 beats

**Gharanas** A particular school of playing

**Jhaptal** A Tal of 10 beats

**Khali** The section of a Kaida which uses band bols.

**Khula** Resonant bols

**Layakari** The relationship between the performed pulse of a composition and the theoretical beat

**Matra** The beat

**Mukhada** A very small phrase or composition ending on a sam

**Pakahawaj** A barrel shaped drum with playing heads on both sides

**Pala** A single phrase of a Tihai

**Palta** A type of variation upon a theme in a Qui'da

**Rag** The entire piece that is played by the melodic instrument in performance of Indian Classical music.

**Sam** The first beat of a cycle

**Sarod** The Sarod is a short-necked, fretless lute carved from a block of teak, with a goat skin sound table. This instrument is said to date from the 19th C

**Sitar** A north-Indian fretted plucked string instrument with a number of melody strings, drone strings and sympathetic strings

**Tal** A particular rhythmic cycle

**Tali** The clapping system of rhythm (Tali = clap, Kali = wave)

**Teental** A common Tal of 16 beats

**Theka** The fundamental rhythmic pattern used for timekeeping

**Tihai** A cadenza composed of three identical sections

**Vibhag** The measure of a bar

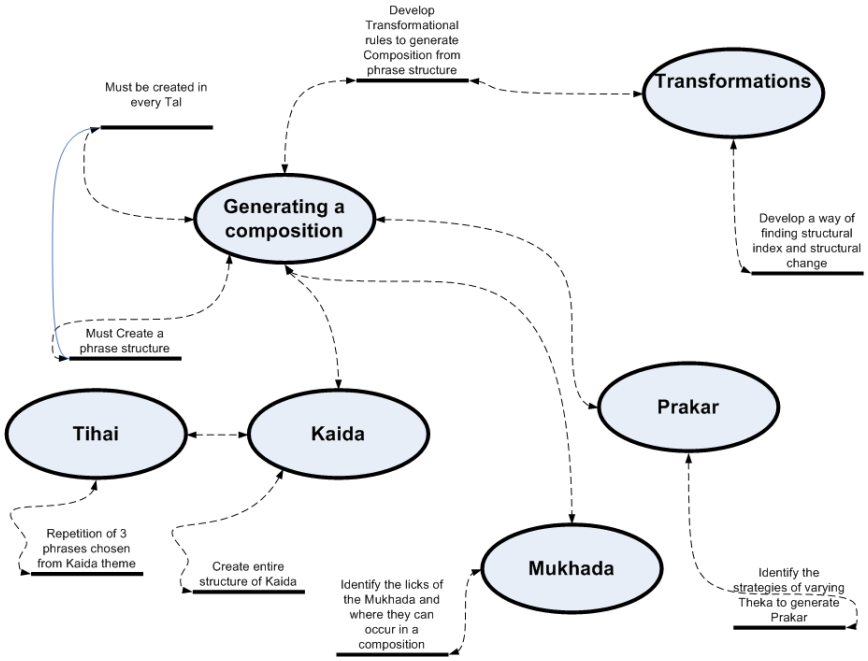
**Vilambit** This section describes the introduction of the Tabla into the piece alongside the melodic instrument.

# Appendix A

# Requirements Elicitation Diagrams

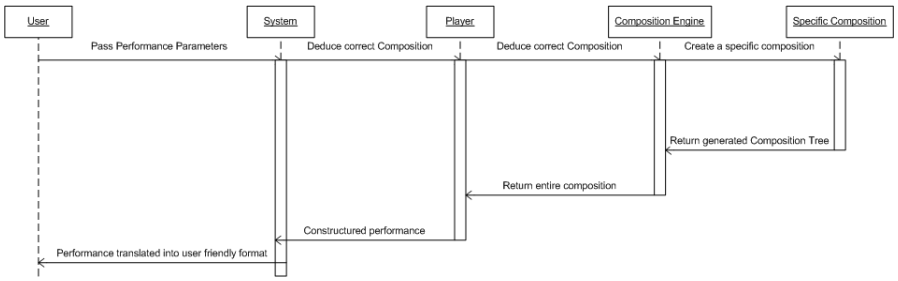
## A.1 Brainstorming

Here is a diagram of the brainstorming session to identify the domain.

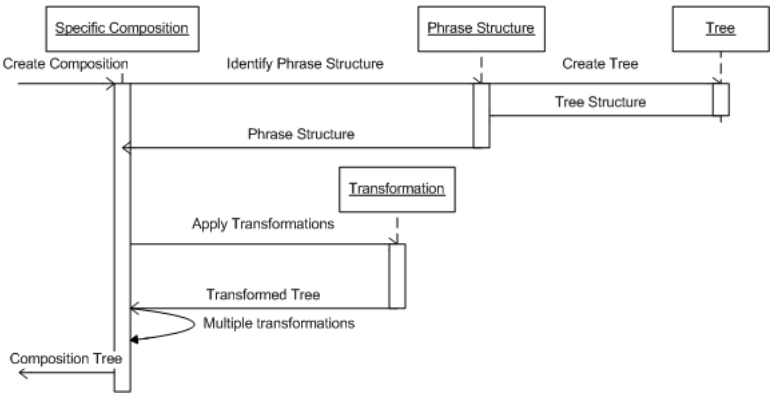



Here are the high level sequence diagrams which were used to identify the main elements in the system and helped derive the main set of functional requirements. These sequence are fairly informal but allow the rapid development on the requirements specification and has aided design.

Initial request to generate a performance:




Specific sequences required in generating a single composition:



# Appendix B

# Full Requirements Specification

**1 Grammatical Modeling**

**1.1 Models and Data Structures**

1.1.1 Develop the basic data structure to create the phrase structure of each composition.

1.1.2 The phrase structure must use a start symbol and production rules

1.1.2.1 The phrase structure grammar must work so that when executed, runs until no more rules can be applied and has constructed a tree

1.1.3 Each of these compositions should have such a phrase structure which enables its generation in any tal

1.1.4 The phrase structure should be easily entered into the system and manipulated

1.1.5 The Tal should be represented in some form

1.1.6 The Tal should be easily entered into the system and manipulated.

1.1.7 Bols should be entered easily into the system and handled to be used in compositions

**1.2 Transformations**

1.2.1 The transformation should take two phases:

1.2.1.1 Structural Index: The function should use Chomsky's 'language' to pattern match against the tree to identify the nodes to be transformed

1.2.1.2 use the same 'language' to perform the transformation

1.2.1.3 transformations will be grouped into four main functions:

1.2.1.3.1 Substitution – replacing nodes

1.2.1.3.2 Permutation – rearranging the nodes of the tree

1.2.1.3.3 Null – make nodes null

1.2.1.3.4 Insertion – creating and inserting new nodes into the tree

**1.3 Compositions**

1.3.1 Each composition should create a phrase structure and understand which bols to use and which tal it is performing in

1.3.2 Each composition should be generated buy use of transformations and tree manipulation with context to the performance

1.3.3 Each composition should record the structure and the layakari of the composition.

1.3.4 Transformational rules should be created either by having fixed rules or to be created on the fly by analysing the composition tree.

1.3.5 Apply each transformational rule to the tree to create the final composition piece.

1.3.6 Parameters to each piece must be provided to tailor the overall composition using the following top level parameters – this may be altered or extended after initial implementation:

1.3.6.1 Mood (Intensity of the piece)

1.3.6.2 Tal

1.3.7 Only basic, theoretical pieces will be generated. There will be a suspension of an artistic model.

1.3.8 The ability affect pieces by artistic influences should be made available.

**1.4 Global Control**

1.4.1 The pieces should be used appropriately according to their purpose/ function.

1.4.1.1 Cadential pieces should be used to stress the end of a piece by building tension and resolving on the Sam.

1.4.1.2 Cyclical pieces should be used for Tabla solo and extending the performance.

1.4.2 High level decisions should be made separate from the development of compositions to choose compositions submitting the parameters for each composition.

1.4.3 The global control system should be able to link the pieces together and have reference to what the pieces are.

**1.5 Graphical User Interface**

1.5.1 The interface must allow the user to input the parameters to the compositions (see 1.3.6)

1.5.2 The user should be able to read the compositions displayed on the interface.

1.5.2.1 Bol words will be used to describe the compositions displayed as text

1.5.2.2 The user should be able to identify the number of bols per beat i.e. the layakari

1.5.2.3 The system should be able to play samples of the bols to play the compositions

1.5.3 The user must be able to select the performance parameters easily and generate the composition by use of a button.

1.5.4 The user should be able to play the composition by pressing a button.

**1.6 System Architecture**

1.6.1 The grammar which generates the language of the compositions should be made modular to the system

1.6.2 The interface should be implemented according to the rules of MVC.

## *Non - Functional Requirements*

**1.7 Performance**

1.7.1 The system should have no memory leaks and memory should be allocated as needed and free appropriately

1.7.2 The program should be relatively small to maintain simplicity and maneuverability

**1.8 Error Handling**

1.8.1 All errors should be handled internally. If any are generated by the user they should inform the user and deliver meaningful messages

**1.9 Usability**

1.9.1 The interface must be simple to use

1.9.2 The output must be easy to read and identifiable

1.9.3 The layout of the buttons and other controls must be clear

**1.10 Cross Platform**

1.10.1 The system should be cross platform

1.10.2 If the system is not cross platform it should be in a state at which it could be refactored easily to be multi-platform

1.10.3 The program must run fully on at least one platform

**1.11 Extendability**

1.11.1 The system should be in a state so that it is easily extended and built upon.

# Appendix C

# Sample Grammar and Output

## C.1 Solo Performance in Teental

### C.1.1 Theka

Phrase Structure Grammar:
$S \rightarrow XXYZ$
$X \rightarrow ABBA$
$Y \rightarrow A(B)(B)(A)$
$Z \rightarrow (A)BBA$
$A \rightarrow Dha$
$B \rightarrow Dhi$
$(A) \rightarrow Ta$
$(B) \rightarrow Ti$

Producing with no bols:
$| A | B | B | A$
$| A | B | B | A$
$| A | (B) | (B) | (A)$
$| (A) | B | B | A$

### C.1.2 Prakar: Valid grammars produced by system

1. $| AA | B | B | A$
   $| AA | B | B | A$
   $| AA | (B) | (B) | (A)$
   $| (A)(A) | B | B | A$

2. $| A | B | B | AA$
   $| A | B | B | AA$
   $| A | (B) | (B) | (A)(A)$
   $| (A) | B | B | AA$

3. If we denote:

   $W1 = TiTa$
   $W2 = TiRa$
   $W3 = KiTa$

   then:
   $| A | B | B | AA$
   $| A | B | B | AA$
   $| A | (B) | (B) | (A)(A)$
   $| (A) | W1 | W2 | W3$

### C.1.3 Mukhada: Valid grammars produced by system

1. If we denote:

   $W1 = TiTa$
   $W2 = KiRa$
   $W3 = NaKa$

   then:
   $| A | B | B | A$

$| A | B | B | A$
$| A | (B) | (B) | (A)$
$| (A) | W1 | W2 | W3$

### C.1.4 Kaida: Valid grammars produced by system

Kaida Phrase Structure Grammar:
$S \rightarrow A1B1A2B2$
$A1 \rightarrow AABC$
$B1 \rightarrow AADE$
$A2 \rightarrow CCBC$
$B2 \rightarrow AAFE$
$A \rightarrow Dha$
$B \rightarrow Ti$
$C \rightarrow Ta$
$D \rightarrow Tu$
$E \rightarrow Na$
$F \rightarrow Dhi$

Using the Nodes:
$| A1 | B1 | A2 | B2$

1st Level variations:

1. $| A1 | A1 | A1 | B1$
   $| A2 | A2 | A2 | B2$

2. $| A1 | B1 | A1 | B2$
   $| A2 | B2 | A2 | B1$

2nd Level Variations:
If we denote:

$A11 = TaDha$
$A12 = DhaTi$
$A21 = TaTa$
$A22 = TaTi$
then:

1. | $A1$ | $B1$ | $A11$ | $A12$ | $B1$
   | $A2$ | $B2$ | $A21$ | $A22$ | $B2$

2. | $A1$ | $A1$ | $B1$ | $B1$
   | $A2$ | $A2$ | $A2$ | $B2$

### C.1.5 Tihai: Valid grammars produced by system

Random bols from Kaida Phrase structure

### C.1.6 Sample Performance using above grammar

**Beginning:**

| $Dha$ | $Dhi$ | $Dhi$ | $DhaDha$
| $Dha$ | $Dhi$ | $Dhi$ | $DhaDha$
| $Dha$ | $Ti$ | $Ti$ | $TaTa$
| $TaSa$ | $TiTa$ | $KiRa$ | $NaKa$

| $Dha$ | $Dhi$ | $Dhi$ | $Dha$
| $Dha$ | $Dhi$ | $Dhi$ | $Dha$
| $Dha$ | $Ti$ | $Ti$ | $Ta$
| $Ta$ | $DhaSa$ | $TiRa$ | $KiTa$

| $Dha$ | $Dhi$ | $Dhi$ | $DhaDha$
| $Dha$ | $Dhi$ | $Dhi$ | $DhaDha$
| $Dha$ | $Ti$ | $Ti$ | $TaTa$
| $Ta$ | $Dhi$ | $Dhi$ | $DhaDha$

| $Dha$ | $Dhi$ | $Dhi$ | $Dha$
| $Dha$ | $Dhi$ | $Dhi$ | $Dha$
| $Dha$ | $Ti$ | $Ti$ | $Ta$
| $Ta$ | $Dhi$ | $Dhi$ | $Dha$

**Kaida Theme**

| $Dha$ | $Dha$ | $Ti$ | $Ta$
| $Dha$ | $Dha$ | $Tu$ | $Na$
| $Ta$ | $Ta$ | $Ti$ | $Ta$
| $Dha$ | $Dha$ | $Dhi$ | $Na$

**Full Speed Kaida**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$

**Variation 1**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TiTa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $TaTa$ | $TiTa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $DhiNa$

**Variation 2**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $TuNa$

**Variation 3**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaDha$ | $DhaTi$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $TaTa$ | $TaTi$ | $DhaDha$ | $DhiNa$

**Variation 4**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaDha$ | $DhaTi$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $TaTa$ | $TaTi$ | $DhaDha$ | $DhiNa$

**Variation 5**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TiTa$
| $DhaDha$ | $TuNa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $TaTa$ | $TiTa$
| $DhaDha$ | $DhiNa$ | $DhaDha$ | $DhiNa$

**Bharan**

| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$
| $DhaDha$ | $TiTa$ | $DhaDha$ | $TuNa$
| $TaTa$ | $TiTa$ | $DhaDha$ | $DhiNa$

**Tihai**

| $DhaNa$ | $TaDha$ | $DhaDha$ | $TiDha$
| $DhaTa$ | $DhaDha$ | $NaTa$ | $DhaDha$
| $DhaTi$ | $DhaDha$ | $TaDha$ | $DhaNa$
| $TaDha$ | $DhaDha$ | $TiDha$ | $DhaTa$
| $Dha$

## C.2 Solo Performance in Jhaptal

### C.2.1 Beginning

| $Dhi$ | $Na$
| $Dhi$ | $Dhi$ | $Na$
| $Ti$ | $Na$
| $Dhi$ | $Dhi$ | $Na$
| $Dhi$ | $Na$
| $Dhi$ | $Dhi$ | $Na$
| $Ti$ | $Na$
| $Dhi$ | $Dhi$ | $Na$

### C.2.2 Kaida Theme

| $DhinGhir$ | $NagTi$
| $TeDhin$ | $GhirNag$ | $TirKit$
| $TinKir$ | $NakTi$
| $TeDhin$ | $GhirNag$ | $TirKit$

### C.2.3 Full Speed Kaida

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$
| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.4 Variation 1

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TiTeDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$
| $TinKirNakTi$ | $TeTinKirNak$
| $TiTeDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$

### C.2.5 Variation 2

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TiTeDhinGhir$ | $NagTirKitDhin$ | $GhirNagTirKit$
| $TinKirNakTi$ | $TeTinKirNak$
| $TiTeDhinGhir$ | $NagTirKitDhin$ | $GhirNagTirKit$

### C.2.6 Variation 3

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TiTeTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$
| $TinKirNakTi$ | $TeTinKirNak$
| $TiTeDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$

### C.2.7 Variation 4

| $TiTeDhinGhir$ | $NagDhinGhirNag$
| $TirKitDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$
| $TiTeTinKir$ | $NakDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.8 Variation 5

| $TiTeDhinGhir$ | $NagDhinGhirNag$
| $TirKitDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$
| $TiTeTinKir$ | $NakDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.9 Variation 6

| $TiTeDhinGhir$ | $NagDhinGhirNag$
| $TirKitDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$
| $TiTeTinKir$ | $NakDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.10 Variation 7

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TiTeDhinGhir$ | $NagTiTeDhin$ | $GhirNagTirKit$
| $TinKirNakTi$ | $TeTinKirNak$
| $TiTeTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.11 Bharan

| $DhinGhirNagTi$ | $TeDhinGhirNag$
| $TirKitTinKir$ | $NakTiTeDhin$ | $GhirNagTirKit$

### C.2.12 Tihai

| $NagTiTinKir$ | $TinKirDhaNag$
| $TiTinKirTin$ | $KirDhaNagTi$ | $TinKirTinKir$
| $Dha$

## C.3 Solo Performance in Rupak

### C.3.1 Beginning

| $TiTi$ | $Ti$ | $Na$
| $DhiDhi$ | $Na$
| $DhiDhi$ | $Na$
| $TiTi$ | $Ti$ | $Na$
| $DhiDhi$ | $Na$
| $DhiDhi$ | $Na$
| $Ti$ | $Ti$ | $NaNa$
| $Dhi$ | $NaNa$
| $Dhi$ | $NaNa$
| $Ti$ | $Ti$ | $Na$
| $Dhi$ | $Na$
| $Dhi$ | $Na$

### C.3.2 Kaida Theme

| $DhaGeTiRaKiTaDhaGeTraKa$ | $DhiNaGiNaDhaGeNaDha$ | $TiRaKiTaDhiTaDhaGeTraKa$
| $DhiNaGiNaTaKeTiRaKiTa$ | $TaGeTraKaTiNaKiNa$

| $DhaGeNaDhaTiRaKiTaDhiTa$ | $DhaGeTrKaDhiNaGiNa$

### C.3.3 Full Speed Kaida

| $DhaGeTiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeNaDha$ | $TiRaKiTaDhiTaDhaGeTraKaDhiNaGiNaTaKeTiRaKiTa$ |
$TaGeTraKaTiNaKiNaDhaGeNaDhaTiRaKiTaDhiTa$
| $DhaGeTrKaDhiNaGiNaDhaGeTiRaKiTaDhaGeTraKa$ | $DhiNaGiNaDhaGeNaDhaTiRaKiTaDhiTaDhaGeTraKa$
| $DhiNaGiNaTaKeTiRaKiTaTaGeTraKaTiNaKiNa$ | $DhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.4 Variation 1

| $DhaGeTiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeDhaGe$ | $TiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeDhaGeTiRaKiTa$ |
$DhaGeTraKaDhiNaGiNaDhaGeNaDhaTiRaKiTaDhiTa$
| $DhaGeTraKaDhiNaGiNaTaKeTiRaKiTaTaGeTraKa$ | $TiNaKiNaDhaGeTaKeTiRaKiTaTaGeTraKaTiNa$
| $KiNaDhaGeDhaGeTiRaKiTaDhaGeTraKaDhiNaGiNa$ | $DhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.5 Variation 2

| $DhaGeTiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeDhaGe$ | $TiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeNaDhaTiRaKiTa$ |
$DhiTaDhaGeTraKaDhiNaGiNaNaDhaTiRaKiTaDhiTa$
| $DhaGeTrKaDhiNaGiNaTaKeTiRaKiTaTaGeTraKa$ | $TiNaKiNaDhaGeTaKeTiRaKiTaTaGeTraKaTiNa$
| $KiNaDhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNa$ | $GiNaNaDhaTiRaKiTaDhiTaDhaGeTraKaDhiNaGiNa$

### C.3.6 Variation 3

| $GeTraKaDhiNaNaDhaTiRaKiTaDhiTaDhaGeTraKaDhi$ | $NaGiNaNaDhaTiRaKiTaDhiTaDhaGeTraKaDhiNaGi$ |
$NaDhaGeTiRaKiTaDhaGiNaDhaGeNaDhaTiRaKiTaDhiTa$
| $DhaGeTraKaDhiNaGiNaGeTraKaTiNaNaDhaTiRa$ | $KiTaDhiTaDhaGeTrKaDhiNaGiNaNaDhaTiRaKiTaDhi$
| $TaDhaGeTrKaDhiNaGiNaTaKeTiRaKiTaTaKiNa$ | $DhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.7 Variation 4

| $GeTraKaDhiNaNaDhaTiRaKiTaDhiTaDhaGeTraKaDhi$ | $NaGiNaNaDhaTiRaKiTaDhiTaDhaGeTraKaDhiNaGi$ |
$NaDhaGeTiRaKiTaDhaGiNaDhaGeNaDhaTiRaKiTaDhiTa$
| $DhaGeTraKaDhiNaGiNaGeTraKaTiNaNaDhaTiRa$ | $KiTaDhiTaDhaGeTrKaDhiNaGiNaNaDhaTiRaKiTaDhi$
| $TaDhaGeTrKaDhiNaGiNaTaKeTiRaKiTaTaKiNa$ | $DhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.8 Variation 5

| $NaDhaTiRaKiTaDhiTaDhaGeTraKaDhiNaGiNaNaDha$ | $TiRaKiTaDhiTaDhaGeTraKaDhiNaGiNaDhaGeTiRaKiTa$ |
$DhaGiNaDhaGeGeTraSDhiNaNaDhaTiRaKiTaDhiTa$
| $DhaGeTraKaDhiNaGiNaNaDhaTiRaKiTaDhiTaDhaGe$ | $TrKaDhiNaGiNaNaDhaTiRaKiTaDhiTaDhaGeTrKa$
| $DhiNaGiNaTaKeTiRaKiTaTaKiNaDhaGeGeTraS$ | $TiNaNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.9 Bharan

| $DhaGeTiRaKiTaDhaGeTraKaDhiNaGiNaDhaGeNaDha$ | $TiRaKiTaDhiTaDhaGeTraKaDhiNaGiNaTaKeTiRaKiTa$ |
$TaGeTraKaTiNaKiNaDhaGeNaDhaTiRaKiTaDhiTa$
| $DhaGeTrKaDhiNaGiNaDhaGeTiRaKiTaDhaGeTraKa$ | $DhiNaGiNaDhaGeNaDhaTiRaKiTaDhiTaDhaGeTraKa$
| $DhiNaGiNaTaKeTiRaKiTaTaGeTraKaTiNaKiNa$ | $DhaGeNaDhaTiRaKiTaDhiTaDhaGeTrKaDhiNaGiNa$

### C.3.10 Tihai

| $KaTraDhaGeDhaNaGiNaGeTrDhiDhiNaTiRaTrDha$ | $TiRaTraNaDhaTaDhaGiDhiNaNaTiRaKaGeTrGeGe$ |
$KaKiTaGiTiRaNaKiTraTraKaNaKaGeTiRaTiRaNaDha$
| $TiRaDhiTiRaTiRaKaNaNaNaTiNaKeNaKiTaNaKiTaNa$ | $TrGeDhaKiTaGeGiDhiTiRaNaNaDhaKaTraDhaGeDha$
| $NaGiNaGeTrDhiDhiNaTiRaTrDhaTiRaTraNaDhaTa$ | $DhaGiDhiNaNaTiRaKaGeTrGeGeKaKiTaGiTiRaNa$
| $Dha$

# Appendix D

# Code Listings

This chapter only shows what was considered the most important classes that have been created. In addition only the cpp files are in this section and not the accompanying header files. To view all code for the system please refer to the accompanying CD for an electronic version.

PlayerObject.cpp

```
/**************PlayetObject.cpp*********************************************
This object is used to control the global structure of the entire performance.
It also translates the performance parameters from the interface into a performance
and adjoins the three sections of the performance together
*****************************************************************************/

#include "PlayerObject.h"

PlayerObject::PlayerObject()
{
        _compGen = new CompositionGenerator();
        CompositionStart = new Node(0,"S");
}

PlayerObject::~PlayerObject()
{
        DestroyTree(CompositionStart);
        CompositionStart = NULL;
}


void PlayerObject::BeginPerformance(Performance _performance, Tal _tal, Mood playerMood)
{
        Node * StartComp;
        Node * MiddleComp;
        Node * EndComp;
        PerformanceMetaData* PerformanceHold;

        //set the mood
        this->_mood = playerMood;

        //Start - CYCLICAL
        PerformanceHold = PerformanceLookup(eSTART, this->_mood, _performance);
        StartComp = _compGen->ChooseComposition(_tal, PerformanceHold->_comp, PerformanceHold->NoOfPhrases);
        StartComp->CompositionName = "----|Beginning|";
        this->CompositionStart->children.push_back(StartComp);
        AdjustMood(PerformanceHold);
        free(PerformanceHold);
```

```
        //Middle - CYCLICAL
        PerformanceHold = PerformanceLookup(eMIDDLE, this->_mood, _performance);
        MiddleComp = _compGen->ChooseComposition(_tal, PerformanceHold->_comp, PerformanceHold->NoOfPhrases);
        this->CompositionStart->children.push_back(MiddleComp);
        AdjustMood(PerformanceHold);
        free(PerformanceHold);

        //End - CADENTIAL
        PerformanceHold = PerformanceLookup(eEND, this->_mood, _performance);
        EndComp = _compGen->ChooseComposition(_tal, PerformanceHold->_comp, PerformanceHold->NoOfPhrases);
        this->CompositionStart->children.push_back(EndComp);
        AdjustMood(PerformanceHold);
        free(PerformanceHold);

}

/*Alters the mood depending on the current mood - only allows the values
between 1 and 3, both numbers inclusive*/

void PlayerObject::ChangeMood(int i)
{
        int currentMood = this->_mood;
        currentMood += i;

        if(currentMood >=3){
                this->_mood = eHEAVY;
        }
        else if(currentMood == 2){
                this->_mood = eMEDIUM;
        }
        else if(currentMood <= 1){
                this->_mood = eLIGHT;
        }
}


/*This method alters the mood dependent on the composition*/
void PlayerObject::AdjustMood(PerformanceMetaData* pmd)
{
        switch(pmd->_comp)
        {
        case eKAIDA:
                switch(pmd->NoOfPhrases)
                {
                case 7:
                        ChangeMood(-2);
                        break;
                case 5:
                        ChangeMood(-1);
                        break;
                case 3:
                        ChangeMood(-1);
                        break;
                }
                break;
        case ePRAKAR:
                switch(pmd->NoOfPhrases)
                {
                case 7:
                        ChangeMood(-1);
                        break;
                case 5:
                        ChangeMood(-1);
                        break;
                case 3:
                        ChangeMood(-1);
                        break;
                case 2:
                        ChangeMood(1);
                        break;
                case 1:
                        ChangeMood(1);
                        break;
                }
        case eTHEKA:
                switch(pmd->NoOfPhrases)
                {
                case 7:
                        ChangeMood(-1);
                        break;
                case 5:
                        ChangeMood(-1);
                        break;
```

```
                case 3:
                        ChangeMood(-1);
                        break;
                case 2:
                        ChangeMood(1);
                        break;
                case 1:
                        ChangeMood(1);
                        break;
                }
                break;
        default:
                ChangeMood(1);
                break;
        }
}

/*Find the composition and size based upon the performance parameters*/
PerformanceMetaData* PlayerObject::PerformanceLookup(CompositionalForm _cf, Mood _md, Performance _p)
{
        PerformanceMetaData* pmd = (PerformanceMetaData*)malloc(sizeof(PerformanceMetaData));

        switch(_p)
        {
        //----SOLO-----------------------------------------------------------------------
        case eSOLO:
                //eSTART
                if(_cf == eSTART && _md == eHEAVY){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 3;
                }
                else if(_cf == eSTART && _md == eMEDIUM){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 2;
                }
                else if(_cf == eSTART && _md == eLIGHT){
                        pmd->_comp = eTHEKA;
                        pmd->NoOfPhrases = 1;
                }

                //eMIDDLE & eEND
                else if(_cf == eMIDDLE && _md == eHEAVY){
                        pmd->_comp = eKAIDA;
                        pmd->NoOfPhrases = 7;
                }
                else if(_cf == eMIDDLE && _md == eMEDIUM){
                        pmd->_comp = eKAIDA;
                        pmd->NoOfPhrases = 5;
                }
                else if(_cf == eMIDDLE && _md == eLIGHT){
                        pmd->_comp = eKAIDA;
                        pmd->NoOfPhrases = 3;
                }
                break;

//----ACCOMPANIST-----------------------------------------------------------------------
        case eACCOMPANIST:
                //eSTART
                if(_cf == eSTART && _md == eHEAVY){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 3;
                }
                else if(_cf == eSTART && _md == eMEDIUM){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 2;
                }
                else if(_cf == eSTART && _md == eLIGHT){
                        pmd->_comp = eTHEKA;
                        pmd->NoOfPhrases = 1;
                }

                //eMIDDLE
                else if(_cf == eMIDDLE && _md == eHEAVY){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 7;
                }
                else if(_cf == eMIDDLE && _md == eMEDIUM){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 5;
                }
                else if(_cf == eMIDDLE && _md == eLIGHT){
                        pmd->_comp = ePRAKAR;
                        pmd->NoOfPhrases = 3;
```

```
                }

                //eEND
                else if(_cf == eEND && _md == eHEAVY){
                        pmd->_comp = eTIHAI;
                        pmd->NoOfPhrases = 1;
                }
                else if(_cf == eEND && _md == eMEDIUM){
                        pmd->_comp = eMUKHADA;
                        pmd->NoOfPhrases = 1;
                }
                else if(_cf == eEND && _md == eLIGHT){
                        pmd->_comp = eMUKHADA;
                        pmd->NoOfPhrases = 1;
                }
                break;

        }
        return pmd;
}


/*Destroy all objects*/
void PlayerObject::DestroyTree(Node* node)
{
        size_t ChildVectorSize;
        int i;

        ChildVectorSize = node->children.size();

        for(i=0;i<ChildVectorSize;i++)
        {
                if(node->children[i] == NULL){
                        delete node;
                        break;
                }
                else{
                        DestroyTree(node->children[i]);
                        delete node;
                        break;
                }
        }
}
```

## CompositionGenerator.cpp

```
/**************CompositionGenerator.cpp*********************************************
This object instantiates the requested composition and produces the size of the
composition by either instantiating it NoOfPhrases times or passes this value to
the specific composition class.
***********************************************************************************/

#include "CompositionGenerator.h"


CompositionGenerator::CompositionGenerator(){}

/*Overloaded constructor*/
CompositionGenerator::CompositionGenerator(Tal _tal, Composition _comp, int NoOfPhrases)
{
        comp = _comp;
        tal = _tal;
        CompositionStart = ChooseComposition(_tal, _comp, NoOfPhrases);
}

//destructor removes pointer to the tree it has generated
CompositionGenerator::~CompositionGenerator()
{
        CompositionStart = NULL;
}

//Main method to create the composition and adjoin compositions of the same type together
Node* CompositionGenerator::ChooseComposition(Tal _tal, Composition _comp, int NoOfPhrases)
{
        Node* root;
        Node* ThekaRoot;
        CompositionObject* compObj;
        PrakarComposition * prakarComp;
        KaidaComposition * kaidaComp;
        MukhadaComposition * mukhadaComp;
        int i;
        Node* root2 = new Node();
```

```
        switch(_comp)
        {
        case eTHEKA:
                for(i=0;i<NoOfPhrases;i++){
                        compObj = new CompositionObject();
                        root = compObj->CreateThekaTree(_tal,true);
                        root2->children.push_back(root);
                }
                break;
        case eMUKHADA:
                for(i=0;i<NoOfPhrases;i++){
                        compObj = new CompositionObject();
                        root = compObj->CreateThekaTree(_tal,true);

                        mukhadaComp = new MukhadaComposition();
                        mukhadaComp->PerformMukhada(root,_tal);
                        root2->children.push_back(root);
                }
                break;
        case ePRAKAR:
                for(i=0;i<NoOfPhrases;i++){
                        prakarComp = new PrakarComposition(_tal);
                        prakarComp->PerformPrakar(prakarComp->PrakarRoot);
                        root = prakarComp->PrakarRoot;
                        root2->children.push_back(root);
                }
                break;
        case eKAIDA:
                kaidaComp = new KaidaComposition();
                kaidaComp->KaidaDriver(_tal, NoOfPhrases);
                root2 = kaidaComp->KaidaStart;
                break;
        }
        return root2;
}
```

## TransformationObject.cpp

```
/**************TransformationComposition.cpp***********************************************
This is the main object that is used to transform the tree using a structual index
and structural change.  It performs the 4 families of transfoamtions - insert,
substitution, nulling and permutation
*******************************************************************************/

#include "TransformationObject.h"

TransformationObject::TransformationObject()
{
}
TransformationObject::~TransformationObject()
{
        int i;
        int SIVectorSize = SIVector.size();
        for(i=0;i<SIVectorSize;i++)
        {
                delete SIVector[i];
        }
        SIVector.clear();
}

void TransformationObject::transformation(string StructuralIndex, string StructuralChange, Node* root, bool prtMatch)
{
        vector<string> structuralIndexVector;
        vector<string> structuralChangeVector;
        vector<Node*> structuralResultVector;
        vector<Node*> structuralParentResultVector;
        int i;
        string::size_type InsertOperator;
        string::size_type NullOperator;
        string::size_type SubstitutionOperator;
        int nodeIndex;
        IndexElement* iE;
        size_t vectorsize1;
        size_t vectorsize2;
        string mystr;
        Node* parentNode;
        Node* childNode;
        int vecPos;
        int intMove = 0;
        string subPos;
```

```
        structuralIndexVector = Split(",",StructuralIndex);
        structuralChangeVector = Split(",",StructuralChange);

        vectorsize1 = structuralChangeVector.size();
        vectorsize2 = structuralChangeVector.size();

        /*IDENTIFY THE NODES OF THE STRUCTURAL INDEX*/
        if(prtMatch){
                TraverseTree(structuralIndexVector, root);
        }
        else{
                //make a copy of the nodes to be altered
                for(i=0;i<vectorsize1;i++)
                {
                        //structuralResultVector.push_back(root->children[i]);
                        iE = (IndexElement*)malloc(sizeof(IndexElement));
                        iE->ChildNode = root->children[i];
                        iE->ParentNode = root;
                        iE->vectorPos = i;
                        this->SIVector.push_back(iE);
                }
        }

        /*PERFORM THE STRUCTURAL CHANGE*/
        //loop through change vector and perform changes
        for(i=0;i<vectorsize2;i++)
        {
                //check if there is an insert with this node
                InsertOperator = structuralChangeVector[i].find("+",0);
                NullOperator = structuralChangeVector[i].find("-",0);
                SubstitutionOperator = structuralChangeVector[i].find("=",0);

                /*SUBSTITUTION TRANSFORMATION*/
                /*if there is a substitution, then do this, overwrite the value and then update the
                structuralchangevector to get rid of the =Arg2 parts and the node can be treated like
                a normal node*/
                if(SubstitutionOperator != string::npos)
                {
                        subPos = parseSubstitution(structuralChangeVector[i], i);
                        structuralChangeVector[i] = subPos;
                }

                /*NULL TRANSFORMATION*/
                //if the node is null, find the corresponding node in the SIVector and make it null from the tree
                if(NullOperator != string::npos)
                {
                        parentNode = this->SIVector[i]->ParentNode;
                        vecPos = this->SIVector[i]->vectorPos;
                        if(vecPos+intMove > parentNode->children.size()){
                                parentNode->children.push_back(childNode);
                        }
                        else{parentNode->children[vecPos+intMove] = NULL;}
                }
                /*INSERT (AND PERMUTATION) TRANSFORMATION*/
                //if there is an insertion with this node, then do it
                else if(InsertOperator != string::npos)
                {
                        intMove = intMove + parseInsert(structuralChangeVector[i], i);
                }
                /*PERMUTATION TRANSFORMATION*/
                else
                {
                        nodeIndex = (atoi(structuralChangeVector[i].c_str())) - 1;
                        if(this->SIVector[i] != NULL || this->SIVector[nodeIndex] != NULL){
                                childNode = this->SIVector[nodeIndex]->ChildNode;
                                parentNode = this->SIVector[i]->ParentNode;
                                //vecPos = this->SIVector[i]->vectorPos;
                                vecPos = i;
                                if(vecPos+intMove > parentNode->children.size()){
                                        parentNode->children.push_back(childNode);
                                }
                                else{parentNode->children[vecPos+intMove] = childNode;}
                        }
                }
        }
        /*After transformation is complete, make the SIVector NULL as the lifetime of the class may not
        end before another transformation and thus wil retain the previous transformation's data and will
        introduce errors if kept*/
        this->SIVector.clear();
}

string TransformationObject::parseSubstitution(string str, int vecPos)
```

```
{
        vector <string> nodeHolder;
        Node* childNode;
        int nodeIndex;
        string insertString;
        string::size_type InsertOperator;

        nodeHolder = Split("=",str);
        nodeIndex = (atoi(nodeHolder[0].c_str())) - 1;
        childNode = this->SIVector[nodeIndex]->ChildNode;

        childNode->value = nodeHolder[1];
        return nodeHolder[0].c_str();
}

void TransformationObject::TraverseTree(vector <string> StructuralIndex, Node* root)
{
        int i = 0;
        int j = 0;
        int k = 0;
        int externalI = 0;

        //i = TraverseTree2(StructuralIndex[0], root, 0);
        for(j=0; j<StructuralIndex.size();j++)
        {
                k = TraverseTree2(StructuralIndex[j], root, i);
                i=k;
        }
}

int TransformationObject::TraverseTree2(string StructuralIndex, Node* root, int start)
{
        int i=0;
        int j=0;
        IndexElement* iE;
        i = start;
        if(i>root->children.size()){return -1;}
        for(i;i<root->children.size();i++)
        {
                if(root->children[i]->value == StructuralIndex)
                {
                        iE = (IndexElement*)malloc(sizeof(IndexElement));
                        iE->ChildNode = root->children[i];
                        iE->ParentNode = root;
                        iE->vectorPos = i;
                        this->SIVector.push_back(iE);
                        return i+1;
                }

                else if(root->children[i] != NULL)
                {
                        j = TraverseTree2(StructuralIndex,root->children[i], 0);
                        if(j==0)
                        {
                                break;
                        }
                        else{
                                return i+1;
                        }
                }
                else
                {
                        return 0;
                }
        }

        //broken out of the loop
        return 0;
}

int TransformationObject::parseInsert(string str, int vecPos)
{
        int i;
        int j;
        int nodeIndex;
        int nodeIndex2;
        int strPlace;
        IndexElement* iE;
        IndexElement* iE_StructuralChange;
        Node* parentNode;
        int ParentLength;
        Node* childNode;
        vector <string> nodeHolder;
```

```
        vector <Node*> tempNodes;
        BolNode* bN;
        vector <Node*> newChildren;

        //seperate out the elements
        nodeHolder = Split("+",str);

        //search through and find the actual node
        for(i=0;i<nodeHolder.size();i++)
        {
                //test to see if element is a number, else
                nodeIndex = (atoi(nodeHolder[i].c_str()));
                if(nodeIndex != 0 && nodeIndex != -1)
                {
                        nodeIndex = nodeIndex-1;
                        strPlace = i;
                        break;
                }
        }

        //We now know where the node is we can insert the elements to it's parent
        //cout << this->SIVector[nodeIndex]->ParentNode->children[1]->value;
        iE = this->SIVector[nodeIndex];
        iE_StructuralChange = this->SIVector[vecPos];
        //we shall save everything in the node's parent's children from the node to the end
        //parentNode = iE->ParentNode;
        //ParentLength = iE->ParentNode->children.size();

        parentNode = iE_StructuralChange->ParentNode;
        ParentLength = iE_StructuralChange->ParentNode->children.size();

        for(j=0;j<ParentLength;j++)
        {
                if(j==iE_StructuralChange->vectorPos)
                {
                        ///******ELEMENTS ON THE LEFT HANDSIDE********/
                        ////Now we shall add (by pushing) each of the new elements left of the node in the nodeHolder
                        ////therefore the elements in the nodeholder upto strPlace
                        for(i=0;i<strPlace;i++)
                        {
                                //first  check whether the insert is another node
                                nodeIndex2 = (atoi(nodeHolder[i].c_str()));
                                if(nodeIndex2 != 0 && nodeIndex2 != -1)
                                {
                                        bN = new BolNode(3,this->SIVector[nodeIndex2-1]->ChildNode->value);
                                }
                                else{bN = new BolNode(3,nodeHolder[i]);}
                                //create the BolNode using the value of the element to be inserted
                                newChildren.push_back(bN);
                                bN->parent = parentNode;
                        }

                        ////Add the original node back
                        newChildren.push_back(iE->ChildNode);

                        ///******ELEMENTS ON THE RIGHT HANDSIDE********/
                        ////Now we shall add (by pushing) each of the new elements right of the node in the nodeHolder
                        ////therefore the elements in the nodeholder after strPlace to the end
                        for(i=strPlace+1;i<nodeHolder.size();i++)
                        {
                                //create the BolNode using the value of the element to be inserted
                                nodeIndex2 = (atoi(nodeHolder[i].c_str()));
                                if(nodeIndex2 != 0 && nodeIndex2 != -1)
                                {
                                        bN = new BolNode(3,this->SIVector[nodeIndex2-1]->ChildNode->value);
                                }
                                else{bN = new BolNode(3,nodeHolder[i]);}
                                //bN = new BolNode(2,nodeHolder[i]);
                                //Add the new BolNode to the node's parents children list
                                newChildren.push_back(bN);
                                bN->parent = parentNode;
                        }
                }
                else
                {
                        newChildren.push_back(parentNode->children[j]);
                }

        }
        parentNode->children.clear();
        parentNode->children = newChildren;
        return (nodeHolder.size() - 1);
}
```

```
void TransformationObject::InsertTransformation(Node* root, int position, vector<Node*> Nodes)
{
        int i;
        vector<Node*> OldNodes;

        //make copies of the nodes children from position until the end (store in vector)
        for(i=position-1;i<root->children.size();i++){
                OldNodes.push_back(root->children[i]);
        }

        //pop off all of the nodes this many times: root->children.size - (position-1)
        while(root->children.size() != (position-1)){
                root->children.pop_back();
        }

        //for each of the elements in vector, add each node to root->children
        for(i=0;i<Nodes.size();i++){
                root->children.push_back(Nodes[i]);
                Nodes[i]->parent = root;
        }

        //for each element in the local vector, add each node to root->children
        for(i=0;i<OldNodes.size();i++){
                root->children.push_back(OldNodes[i]);
                OldNodes[i]->parent = root;
        }
}
```

## CompositionObject.cpp

```
/**************CompositionObject.cpp***********************************************
This object is the main parent class to all compositions that are generated.
It includes the main phrase structure grammar to build the Tal currently Teental,
Jhaptal and Rupak - therefore creates the main Theka composition.It also contains
the Bag of Tricks knowledgebase which are selected bols held in a tree structure.

This objetc also contains the other Fixed Transformations: AppendBolsOfTheme,
AppendBolsToTreeEnd and ChangeLayakari.
*******************************************************************************/

#include "CompositionObject.h"

CompositionObject::CompositionObject(){}
CompositionObject::~CompositionObject(){}

/*This method uses the phrase grammar for the specific Tal and generates the
tree structure by using the adjoin function*/

Node* CompositionObject::CreateThekaTree(Tal _tal, bool withBols)
{
        Node* StartNode = NULL;
        phrase * theka = new phrase();
        int i;
        size_t ChildVectorSize;
        switch(_tal)
        {
        case eTEENTAL:
         theka->addRule("S",0,"X,X,Y,Z",1);
         theka->addRule("Y",1,"A,(B),(B),(A)",2);
         theka->addRule("X",1,"A,B,B,A",2);
         theka->addRule("Z",1,"(A),B,B,A",2);
         if(withBols){
                theka->addBolRule("A",2,"Dha",3);
                theka->addBolRule("B",2,"Dhi",3);
                theka->addBolRule("(A)",2,"Ta",3);
                theka->addBolRule("(B)",2,"Ti",3);
         }
         StartNode = AdjoinTree(theka);
         break;
        case eJHAPTAL:
         theka->addRule("S",0,"X,Y,Z,Y",1);
         theka->addRule("X",1,"A,B",2);
         theka->addRule("Y",1,"A,A,B",2);
         theka->addRule("Z",1,"(A),B",2);
         if(withBols){
                theka->addBolRule("A",2,"Dhi",3);
                theka->addBolRule("B",2,"Na",3);
                theka->addBolRule("(A)",2,"Ti",3);
         }
         StartNode = AdjoinTree(theka);
         break;
```

```
        case eRUPAKTAL:
         theka->addRule("S",0,"X,Y,Y",1);
         theka->addRule("X",1,"A,A,B",2);
         theka->addRule("Y",1,"C,B",2);
         if(withBols){
                 theka->addBolRule("A",2,"Ti",3);
                 theka->addBolRule("B",2,"Na",3);
                 theka->addBolRule("C",2,"Dhi",3);
         }
         StartNode = AdjoinTree(theka);
         break;
        }

        /*If the theka has been created with no bols, then the leaf nodes (type 2)
        have not been properly duplicated*/
        if(!withBols){
                ChildVectorSize = StartNode->children.size();
                for(i=0;i<ChildVectorSize; i++)
                {
                        //find any children which are normal nodes
                        CreateNodeObjects(StartNode->children[i]);
                }
        }
        return StartNode;
}

//Essentially hardcoded pieces of bols that are common and can be used is
//various compositions and tals for increasing bols density and style
Node* CompositionObject::BagOfTricks(int NoOfBols)
{
        int i;
        Node* StartNode = NULL;
        phrase * bols = new phrase();

        switch(NoOfBols){
                case 0:
                        //Ti Ra Ki Ta
                        bols->addRule("S",0,"A,B,C,D",2);
                        bols->addBolRule("A",2,"Ti",3);
                        bols->addBolRule("B",2,"Ra",3);
                        bols->addBolRule("C",2,"Ki",3);
                        bols->addBolRule("D",2,"Ta",3);
                        break;
                case 1:
                        //Dha - Ti Ra Ki Ta
                        bols->addRule("S",0,"A,B,C,D,E,F",2);
                        bols->addBolRule("A",2,"Dha",3);
                        bols->addBolRule("B",2,"Sa",3);
                        bols->addBolRule("C",2,"Ti",3);
                        bols->addBolRule("D",2,"Ra",3);
                        bols->addBolRule("E",2,"Ki",3);
                        bols->addBolRule("F",2,"Ta",3);
                        break;
                case 2:
                        //Ti Ra Ki Ta Ta Ka Ta
                        bols->addRule("S",0,"A,B,C,D,D,E,D",2);
                        bols->addBolRule("A",2,"Ti",3);
                        bols->addBolRule("B",2,"Ra",3);
                        bols->addBolRule("C",2,"Ki",3);
                        bols->addBolRule("D",2,"Ta",3);
                        bols->addBolRule("E",2,"Ka",3);
                        break;
                case 3:
                        //Ta - Ti Ta Ki Ra Na Ka
                        bols->addRule("S",0,"A,B,C,A,D,E,F,G",2);
                        bols->addBolRule("A",2,"Ta",3);
                        bols->addBolRule("B",2,"Sa",3);
                        bols->addBolRule("C",2,"Ti",3);
                        bols->addBolRule("D",2,"Ki",3);
                        bols->addBolRule("E",2,"Ra",3);
                        bols->addBolRule("F",2,"Na",3);
                        bols->addBolRule("G",2,"Ka",3);
                        break;
        }

        StartNode = AdjoinTree(bols);
        return StartNode;
}

/*Recursive Descent Phrase Structure Method*/
Node* CompositionObject::AdjoinTree(phrase* phraseObject)
{
        Node* StartNode;
```

```
        int i;
        size_t ChildVectorSize;
        //first find the S - start symbol - iterate through phrase vector list
        StartNode = FindRuleInPhrase(phraseObject,"S");
        AdjoinTree2(phraseObject,StartNode);
        ChildVectorSize = StartNode->children.size();

        return StartNode;

}

/*This method uses another function to iterate through a production rules
RHS symbols and for each element, find the corresponding rule in the phrase
object passed*/
void CompositionObject::AdjoinTree2(phrase* phraseObject, Node* ruleObject)
{
        int i;
        Node* result;
        Node* resultCopy;
        Node* tmp;
        size_t ChildVectorSize;

        ChildVectorSize = ruleObject->children.size();

        for(i=0;i<ChildVectorSize; i++)
        {
                resultCopy = FindRuleInPhrase2(phraseObject,ruleObject->children[i]->value);
                if(resultCopy != NULL){
                        result = new Node(resultCopy);
                        ruleObject->children[i] = result;
                        result->parent = ruleObject;
                        /*if this node's children are Bols, then each of them need to be made into seperate objects
                        NO SHALLOW COPYING OF BOLS*/
                        CreateBolObjects(result);
                }
        }
}

void CompositionObject::CreateBolObjects(Node* root)
{
        int i;
        int childSize;
        Node* nN;
        BolNode* bN;

        childSize = root->children.size();
        for(i=0;i<childSize;i++)
        {
                if(root->children[i]->type == 3)
                {
                        bN = new BolNode(3,root->children[i]->value);
                        root->children[i] = bN;
                }
        }
}
void CompositionObject::CreateNodeObjects(Node* root)
{
        int i;
        int childSize;
        Node* nN;

        childSize = root->children.size();
        for(i=0;i<childSize;i++)
        {
                if(root->children[i]->type == 2)
                {
                        nN = new Node(1,root->children[i]->value);
                        root->children[i] = nN;
                        nN->parent = root;
                }
                else{CreateNodeObjects(root->children[i]);}
        }
}

Node* CompositionObject::FindRuleInPhrase2(phrase* phraseObject, string strValue)
{
        int i;
        int j;
    int N;
        vector <Node*> RuleHolder;
        Node* tempNode = NULL;
        size_t PhraseVectorSize;
```

```
        PhraseVectorSize = phraseObject->ListBegin.size();
        for(i=0;i<PhraseVectorSize;i++)
        {
                if(phraseObject->ListBegin[i]->value == strValue)
                {
                        //For each rule found put into a vector
                        RuleHolder.push_back(phraseObject->ListBegin[i]);
                }
        }

        /*From the Ruleholder choose one, for now at random
        and assign to tempNode*/
        if(RuleHolder.empty() == false){
                N = RuleHolder.size();
                j = rand() % N;
                tempNode = RuleHolder[j];
        }

        if(tempNode == NULL)
        {
                return NULL;
        }
        else
        {
                AdjoinTree2(phraseObject, tempNode);
                return tempNode;
        }
}

/*Finds a specific rule in the phrase object*/
Node* CompositionObject::FindRuleInPhrase(phrase* phraseObject, string strValue)
{
        int i;
        Node* tempNode;
        size_t PhraseVectorSize;

        PhraseVectorSize = phraseObject->ListBegin.size();
        for(i=0;i<PhraseVectorSize;i++)
        {
                if(phraseObject->ListBegin[i]->value == strValue)
                {
                        tempNode = phraseObject->ListBegin[i];
                        break;
                }
        }
        return tempNode;
}


/*This method iterates through the main phrase structure tree starting at the end and
working backards adding bols from the vector passed to it at the appropriate layakari*/
void CompositionObject::AppendBolsToTreeEnd(vector<string> Bols,Node* root, int layakari)
{
        int laya = layakari;
        int BolsTraker = Bols.size()-(laya);
        int BolsCount;
        int BolsPerBeat;
        BolNode* bN;
        int BeatCount;
        bool track = true;
        bool AppendFinished = false;

        //must iterate through the sentences and the beats of the tree (starting at the end)
        int SentenceCount = root->children.size()-1;

        for(SentenceCount;SentenceCount>=0;SentenceCount--){
                BeatCount=root->children[SentenceCount]->children.size()-1;
                for(BeatCount;BeatCount>=0;BeatCount--){

                        //clear all of the children and then we shall append new bols
                        if(BolsTraker >=0 && track == true){
                                root->children[SentenceCount]->children[BeatCount]->children.clear();
                                track = false;

                                for(BolsCount=BolsTraker;(BolsCount<BolsTraker+laya && BolsTraker >=0);BolsCount++){

                                        bN = new BolNode(3,Bols[BolsCount]);
                                        root->children[SentenceCount]->children[BeatCount]->children.push_back(bN);
                                        root->children[SentenceCount]->children[BeatCount]->type = 2;
                                }
                        }
                        BolsTraker = BolsTraker -(laya);
                        if(BolsTraker <= 0 && AppendFinished == false)
```

```
                        {
                                AppendFinished = true;
                                laya = BolsTraker + laya;
                                BolsTraker = 0;
                        }
                        track = true;
                }

        }
}

//replaces the roots children bols with that of the bols in the vector
Node* CompositionObject::ChangeLayakari(vector<string> Bols, Tal _tal, int layakari, bool BolLengthOnly)
{
        //use the tree from root to create the new tree of the specified layakari
        Node* ThekaRoot;
        int ThekaIndex;
        int newThekaSentence;
        int newThekaBeat;
        vector <string> BolsPerBeat;
        int BolsIndex = 0;
        int BolsMax;
        int BolsIndexTemp=0;
        BolNode* bN;
        int i;
        int NoOfBolsPerBeat;


        //Create the Tal phrase structure with no bols
        ThekaRoot = CreateThekaTree(_tal, false);

    /*we can now iterate+loop through the vector of bols and create bols and attach to
        each beat of the ThekaRoot of the size of the layakari*/

        //iterate through the sentence
        for(newThekaSentence=0;newThekaSentence<ThekaRoot->children.size();newThekaSentence++)
        {
                //iterate through each beat
                for(newThekaBeat=0;newThekaBeat<ThekaRoot->children[newThekaSentence]->children.size();newThekaBeat++)
                {
                        //iterate through the vector of amount of layakari (must minus 1 as it is a vector index)
                        BolsMax = (BolsIndex+layakari)-1;
                        for(BolsIndex;BolsIndex<=BolsMax;BolsIndex++)
                        {

                                //If we have reached the end of the vector, then start at beginning again

                                if(BolsIndex>Bols.size()-1)
                                {
                                        //Only do upto the number of bols that are given
                                        if(BolLengthOnly == true)
                                        {
                                                Bols.clear();
                                                cleanTree(ThekaRoot);
                                                return ThekaRoot;
                                        }
                                        BolsIndex = 0; BolsMax = ((BolsMax-(Bols.size()-1)) -1);
                                }
                                //Create the new Bol

                                //Must analyse the string
                                BolsPerBeat = Split("+",Bols[BolsIndex]);
                                for(NoOfBolsPerBeat=0;NoOfBolsPerBeat<BolsPerBeat.size();NoOfBolsPerBeat++)
                                {
                                        //bN = new BolNode(3,BolsOfTree[BolsIndex]);
                                        bN = new BolNode(3,BolsPerBeat[NoOfBolsPerBeat]);
                                        //append to tree
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->children.push_back(bN);
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->type = 2;

                                }
                        }
                }
        }

        return ThekaRoot;

}

Node* CompositionObject::ChangeLayakari(Node* _root, Tal _tal, int layakari, bool BolLengthOnly)
{
        //use the tree from root to create the new tree of the specified layakari
        Node* ThekaRoot;
```

```
        int ThekaIndex;
        int newThekaSentence;
        int newThekaBeat;
        vector <string> BolsPerBeat;
        int BolsIndex = 0;
        int BolsMax;
        int BolsIndexTemp=0;
        BolNode* bN;
        int i;
        int NoOfBolsPerBeat;

        //Collect the bols of the tree
        this->BolsOfTree.clear();
        CollectBols(_root);

        //Create the Tal phrase structure with no bols
        ThekaRoot = CreateThekaTree(_tal, false);

    /*we can now iterate+loop through the vector of bols and create bols and attach to
        each beat of the ThekaRoot of the size of the layakari*/

        //iterate through the sentence
        for(newThekaSentence=0;newThekaSentence<ThekaRoot->children.size();newThekaSentence++)
        {
                //iterate through each beat
                for(newThekaBeat=0;newThekaBeat<ThekaRoot->children[newThekaSentence]->children.size();newThekaBeat++)
                {
                        //iterate through the vector of amount of layakari (must minus 1 as it is a vector index)
                        BolsMax = (BolsIndex+layakari)-1;
                        for(BolsIndex;BolsIndex<=BolsMax;BolsIndex++)
                        {

                                //If we have reached the end of the vector, then start at beginning again

                                if(BolsIndex>BolsOfTree.size()-1)
                                {
                                        //Only do upto the number of bols that are given
                                        if(BolLengthOnly == true)
                                        {
                                                this->BolsOfTree.clear();
                                                cleanTree(ThekaRoot);
                                                return ThekaRoot;
                                        }
                                        BolsIndex = 0; BolsMax = ((BolsMax-(BolsOfTree.size()-1)) -1);
                                }
                                //Create the new Bol

                                //Must analyse the string
                                BolsPerBeat = Split("+",BolsOfTree[BolsIndex]);
                                for(NoOfBolsPerBeat=0;NoOfBolsPerBeat<BolsPerBeat.size();NoOfBolsPerBeat++)
                                {
                                        //bN = new BolNode(3,BolsOfTree[BolsIndex]);
                                        bN = new BolNode(3,BolsPerBeat[NoOfBolsPerBeat]);
                                        //append to tree
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->children.push_back(bN);
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->type = 2;
                                }
                        }
                }
        }
        this->BolsOfTree.clear();
        return ThekaRoot;
}

Node* CompositionObject::AppendVectorOfBols(vector<string>Bols, Tal _tal, int layakari)
{
        Node* start = new Node(0,"t");
        Node* temp1;
        Node* temp2;
        vector<string> tempBols;
        int i=0;
        int j=0;

        while(i<(Bols.size()))
        {
                j=i+(_tal*layakari);
                if(j>Bols.size()){
                        break;
                }
                tempBols.clear();
                for(i;i<j;i++)
                {
                        tempBols.push_back(Bols[i]);
```

```
                }

                temp1 = ChangeLayakari(tempBols, _tal, layakari, true);
                start->children.push_back(temp1);
                i=j;
        }
        return start;
}

Node* CompositionObject::AppendThemeOfBols(vector<string>Bols, Tal _tal, int layakari)
{
        Node* ThekaRoot;
        Node* temp;
        int ThekaIndex;
        int newThekaSentence;
        int newThemeSentence;
        int newThekaBeat;
        vector<string> BolsPerBeat;
        int BolsPerSentence;
        int NoOfBolsPerBeat;
        int BolsIndex = 0;
        int BolsMax;
        int BolsIndexTemp=0;
        BolNode* bN;
        int i;
        int i2;

        //Create the Tal phrase structure with no bols
        ThekaRoot = CreateThekaTree(eTEENTAL, false);
        //Get how many bols per sentence (rhyming sentence each is)
        BolsPerSentence = CalculateQuadraticStructure(Bols, _tal);

        for(i=0;i<ThekaRoot->children.size();i++)
        {
                ThekaRoot->children[i]->children.clear();
                for(i2=0;i2<BolsPerSentence;i2++)
                {
                        temp = new Node(2,"temp");
                        ThekaRoot->children[i]->children.push_back(temp);
                }
        }

    /*we can now iterate+loop through the vector of bols and create bols and attach to
        each beat of the ThekaRoot of the size of the layakari*/

        //iterate through the sentence
        for(newThekaSentence=0;newThekaSentence<ThekaRoot->children.size();newThekaSentence++)
        {
                //iterate through each beat
                for(newThekaBeat=0;newThekaBeat<ThekaRoot->children[newThekaSentence]->children.size();newThekaBeat++)
                {

                        //iterate through the vector of amount of layakari (must minus 1 as it is a vector index)
                        BolsMax = (BolsIndex+layakari)-1;
                        for(BolsIndex;BolsIndex<=BolsMax;BolsIndex++)
                        {
                                //If we have reached the end of the vector, then start at beginning again
                                if(BolsIndex>Bols.size()-1)
                                {
                                        //If there are any non
                                        cleanTree(ThekaRoot);
                                        return ThekaRoot;
                                }
                                //Must analyse the string
                                BolsPerBeat = Split("+",Bols[BolsIndex]);

                                //for each of the bols found, append to the beat
                                for(NoOfBolsPerBeat=0;NoOfBolsPerBeat<BolsPerBeat.size();NoOfBolsPerBeat++)
                                {
                                        //Create the new Bol
                                        bN = new BolNode(3,BolsPerBeat[NoOfBolsPerBeat]);
                                        //append to tree
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->children.push_back(bN);
                                        ThekaRoot->children[newThekaSentence]->children[newThekaBeat]->type = 2;
                                        bN->parent = ThekaRoot->children[newThekaSentence]->children[newThekaBeat];
                                }
                        }
                }
        }

        return ThekaRoot;
}
```

```
/*This method calculates the number of themes there are in a Theme*/
int CompositionObject::CalculateQuadraticStructure(vector<string>Bols, Tal _tal)
{
        int BolsPerSentence;
        int NoOfBolsInTheme = Bols.size();

        //If bols of theme = bols of tal and is 6, then BolsPerSentence = 3
        if(NoOfBolsInTheme == eDADRATAL)
        {
                BolsPerSentence = 3;
        }
        else if(NoOfBolsInTheme % 4 == 0){
                BolsPerSentence = NoOfBolsInTheme / 4;
                return BolsPerSentence;
        }
        else{
                //This is an irregular tal (7-Rhupak/9-Matta/11/13/15...)
                BolsPerSentence = 0;
        }

        return BolsPerSentence;

}

/*This method is used to collect the bols off a tree and append to the public
propety vector as it works recursively*/
void CompositionObject::CollectBols(Node* node)
{
    size_t ChildVectorSize;
        int i;
        int j;

        string BolsPerBeat = "";
        ChildVectorSize = node->children.size();
        for(i=0;i<ChildVectorSize;i++)
        {
                if(node->children[i] != NULL){
                        if(node->children[i]->type == 3)
                        {
                                BolsPerBeat += node->children[i]->value;
                                BolsPerBeat +="+";

                        }
                        else
                        {
                                CollectBols(node->children[i]);
                        }
                }

                if(BolsPerBeat != ""){
                        BolsPerBeat = BolsPerBeat.substr(0,BolsPerBeat.length()-1);
                        BolsOfTree.push_back(BolsPerBeat);
                        BolsPerBeat = "";
                }

        }
}


Node* CompositionObject::copyTree(Node* root)
{
        //iterate through tree and copy each node

        //copy the parent and then do each for children
        int i;
        Node* newChild;
        BolNode* newBol;

        Node* newRoot = new Node(root->type,root->value);

        for(i=0;i<root->children.size();i++)
        {
                //recurse for children
                if(root->children[i]->children.empty() == false)
                {
                        newChild = copyTree(root->children[i]);
                        newRoot->children.push_back(newChild);
                        newChild->parent = newRoot;
                }
                else if(root->children[i]->type == 3)
                {
                        newBol = new BolNode(3,root->children[i]->value);
                        newRoot->children.push_back(newBol);
```

```
                        //newRoot->type = 2;
                        newBol->parent = newRoot;
                }
                else
                {
                        newChild = new Node(root->type,root->children[i]->value);
                        newRoot->children.push_back(newChild);
                        newChild->parent = newRoot;
                }

        }

        return newRoot;
}

void CompositionObject::cleanTree(Node* root)
{
        //for each of the root's children, recurse to the bottom of the branch
        //and if there are no bol leaf nodes then delete
        int i;

        for(i=root->children.size()-1;i>=0;i--)
        {
                if(!FindBolLeaf(root->children[i])){
                        root->children.pop_back();
                }
        }

}

bool CompositionObject::FindBolLeaf(Node* root)
{
        bool result;

        if(root->type == 3)
        {
                return true;
        }
        else
        {
                if(root->children.empty() == true){
                        return false;
                }
                else{
                        result = FindBolLeaf(root->children[0]);
                        return result;
                }
        }
}
```

## PrakarComposition.cpp

```
/**************PrakarComposition.cpp*********************************************
THis uses the three strategies of either doubling the bols of the first or last beat
or uses a set of bols from the bag of tricks.  There is also another method which uses a
combination of two strategies
********************************************************************************/

#include "PrakarComposition.h"

PrakarComposition::PrakarComposition(Tal _tal)
{
        //from the root, the first children are the sentences of the taal
        //their children are the beats of the sentence
        transformObj = new TransformationObject();
        PrakarRoot = CreateThekaTree(_tal,true);
        SetUpPrakarStyles();

}
PrakarComposition::~PrakarComposition(){}

void PrakarComposition::PerformPrakar(Node* ThekaRoot)
{
        int StyleValue;

        //linear stochastic process
        StyleValue = (int)linrand(4);
        (this->*PrakarFuncArr[StyleValue])(ThekaRoot);
}

/*Setting up the strategies as function pointers - this is an experimental technique
to see if the method of stochastically choosing a function from a vector of
```

```
function pointers ordered in increasing bol density*/
void PrakarComposition::SetUpPrakarStyles()
{
   PrakarFuncArr[0] = &PrakarComposition::PrakarStyle4;
   PrakarFuncArr[1] = &PrakarComposition::PrakarStyle3;
   PrakarFuncArr[2] = &PrakarComposition::PrakarStyle2;
   PrakarFuncArr[3] = &PrakarComposition::PrakarStyle1;
}

/*DOuble the bols of the first beat*/
void PrakarComposition::PrakarStyle1(Node* ThekaRoot)
{
        int sentence;
        int No_of_Sentences;
        Node* beat;

        No_of_Sentences = ThekaRoot->children.size();
        //iterate through each of the sentences
        for(sentence=0;sentence<No_of_Sentences;sentence++)
        {
                //for each sentence, copy the first bol and add to first beat
                beat = ThekaRoot->children[sentence]->children[0];
                transformObj->transformation("1","1+1",beat,false);
        }
}

/*DOuble the bols of the last beat*/
void PrakarComposition::PrakarStyle2(Node* ThekaRoot)
{
        int sentence;
        int No_of_Sentences;
        int No_of_bols;
        Node* beat;

        No_of_Sentences = ThekaRoot->children.size();
        //iterate through each of the sentences
        for(sentence=0;sentence<No_of_Sentences;sentence++)
        {
                //for each sentence, copy the first bol and add to first beat
                No_of_bols = ThekaRoot->children[sentence]->children.size();
                beat = ThekaRoot->children[sentence]->children[No_of_bols-1];
                transformObj->transformation("1","1+1",beat,false);
        }
}
/*use the bols from the bag of tricks*/
void PrakarComposition::PrakarStyle3(Node* ThekaRoot)
{
        Node* BoT;
        int BoTType = (int)linrand(4);
        //We get a tree of bols from our bag of tricks
        BoT = BagOfTricks(BoTType);
        this->BolsOfTree.clear();
        CollectBols(BoT);

        //append them to the tree
        AppendBolsToTreeEnd(this->BolsOfTree,ThekaRoot,2);

}

/*Use two strategies*/
void PrakarComposition::PrakarStyle4(Node* ThekaRoot)
{
        // pick 2 arbitary styles and then do them both
        int StyleValue1;
        int StyleValue2;

        //linear stochastic process
        StyleValue1 = (int)linrand(3);
        StyleValue2 = (int)linrand(3);

        (this->*PrakarFuncArr[StyleValue1])(ThekaRoot);
        (this->*PrakarFuncArr[StyleValue2])(ThekaRoot);

}
```

## MukhadaComposition.cpp

```
/***************MukhadaComposition.cpp************************************************
This is a simple class which generates the Mukhada.  It uses the two strategies
of either appending bols from the Bag of Tricks or using a Kaida theme to
the end of a phrase structure tree
*************************************************************************/
```

```
#include "MukhadaComposition.h"

MukhadaComposition::MukhadaComposition(){}
MukhadaComposition::~MukhadaComposition(){}

void MukhadaComposition::PerformMukhada(Node* root, Tal _tal)
{
        Node* BolsOfMukhada;
        KaidaComposition* kaidaComp;
        int MukhadaType = (int)linrand(2);
        int BoTType = (int)linrand(3);
        int ThemeNumber;
        int speed1;

        //We get a tree of bols from our bag of tricks
        //This will vary depending on the mood

        //1) Bag of tricks or Kaida Theme
        //0 = BoT, 1 = Kaida
        switch(MukhadaType)
        {
                case 1:
                        BolsOfMukhada = BagOfTricks(BoTType);
                        speed1 = 2;
                        break;
                case 0:
                        kaidaComp = new KaidaComposition();
                        ThemeNumber = kaidaComp->ChooseKaidaTheme(_tal);
                        BolsOfMukhada = kaidaComp->GetKaidaTheme(_tal,ThemeNumber);
                        speed1 = 4;
                        break;
        }

        this->BolsOfTree.clear();
        CollectBols(BolsOfMukhada);

        //2) speed

        //append them to the tree
        AppendBolsToTreeEnd(this->BolsOfTree,root,speed1);
}
```

## KaidaComposition.cpp

```
/**************KaidaComposition.cpp***********************************************
This classes generates the entire of the Kaida composition.  It uese the Tal that
is passed to select one of the Themes for that Tal.  There can be numerous themes
for a Tal - controlled by the enum KaidaThemeEnum.
The NoOfPhrases value that is passed is used to generate the number of variations
*********************************************************************************/

#include "KaidaComposition.h"

KaidaComposition::KaidaComposition()
{
        transformObj = new TransformationObject();
}
KaidaComposition::~KaidaComposition()
{
        transformObj = NULL;
        KaidaStart = NULL;
}

/*This uses the phrase structure of the Kaida theme to create the tree*/
Node* KaidaComposition::GetKaidaTheme(Tal _tal, int KaidaNumber)
{
        //For each kaida we create a new phrase
        Node* KaidaThemeRoot;
        Node* KaidaFastRoot;
        phrase * kaidaTheme1 = new phrase();
        vector<string> kaidaTheme_vector;

        switch(_tal){
                case eTEENTAL:
                switch(KaidaNumber)
                {
                case 0:
                        kaidaTheme1->addRule("S",0,"W,X,Y,Z",1);
                        kaidaTheme1->addRule("W",1,"A,A,B,C",2);
                        kaidaTheme1->addRule("X",1,"A,A,D,E",2);
                        kaidaTheme1->addRule("Y",1,"C,C,B,C",2);
```

```
            kaidaTheme1->addRule("Z",1,"A,A,F,E",2);
            kaidaTheme1->addBolRule("A",2,"Dha",3);
            kaidaTheme1->addBolRule("B",2,"Ti",3);
            kaidaTheme1->addBolRule("C",2,"Ta",3);
            kaidaTheme1->addBolRule("D",2,"Tu",3);
            kaidaTheme1->addBolRule("E",2,"Na",3);
            kaidaTheme1->addBolRule("F",2,"Dhi",3);
            KaidaThemeRoot = AdjoinTree(kaidaTheme1);
            this->BolsOfTree.clear();
            CollectBols(KaidaThemeRoot);
            KaidaFastRoot = AppendThemeOfBols(this->BolsOfTree,_tal,1);
            //KaidaFastRoot = ChangeLayakari(KaidaThemeRoot,_tal,2,false);
            break;
    case 1:
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Dha");

            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Dha");

            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Ge");

            kaidaTheme_vector.push_back("Tin");
            kaidaTheme_vector.push_back("Na");
            kaidaTheme_vector.push_back("Kin");
            kaidaTheme_vector.push_back("Na");

            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ta");

            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ta");

            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Ge");

            kaidaTheme_vector.push_back("Dhin");
            kaidaTheme_vector.push_back("Na");
            kaidaTheme_vector.push_back("Gin");
            kaidaTheme_vector.push_back("Na");

            KaidaFastRoot = AppendThemeOfBols(kaidaTheme_vector,_tal,1);
            break;
    case 2:
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");

            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Tu");
            kaidaTheme_vector.push_back("Na");

            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ta");
            kaidaTheme_vector.push_back("Ti");
            kaidaTheme_vector.push_back("Ta");

            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Dha");
            kaidaTheme_vector.push_back("Dhin");
            kaidaTheme_vector.push_back("Na");

            KaidaFastRoot = AppendThemeOfBols(kaidaTheme_vector,_tal,1);
            break;
    }
    break;
    case eJHAPTAL:
            switch(KaidaNumber)
```

```
        {
        case 0:
                kaidaTheme_vector.push_back("Dhin");
                kaidaTheme_vector.push_back("Ghir");
                kaidaTheme_vector.push_back("Nag");
                kaidaTheme_vector.push_back("Ti");
                kaidaTheme_vector.push_back("Te");

                kaidaTheme_vector.push_back("Dhin");
                kaidaTheme_vector.push_back("Ghir");
                kaidaTheme_vector.push_back("Nag");
                kaidaTheme_vector.push_back("Tir");
                kaidaTheme_vector.push_back("Kit");

                kaidaTheme_vector.push_back("Tin");
                kaidaTheme_vector.push_back("Kir");
                kaidaTheme_vector.push_back("Nak");
                kaidaTheme_vector.push_back("Ti");
                kaidaTheme_vector.push_back("Te");

                kaidaTheme_vector.push_back("Dhin");
                kaidaTheme_vector.push_back("Ghir");
                kaidaTheme_vector.push_back("Nag");
                kaidaTheme_vector.push_back("Tir");
                kaidaTheme_vector.push_back("Kit");

                KaidaFastRoot = AppendThemeOfBols(kaidaTheme_vector,_tal,1);
                //(kaidaTheme_vector.size() / _tal)
                break;
        }
        break;
case eRUPAKTAL:
        switch(KaidaNumber)
        {
                case 0:
                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("Ge");
                kaidaTheme_vector.push_back("TiRa");
                kaidaTheme_vector.push_back("KiTa");

                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("Ge");
                kaidaTheme_vector.push_back("Tra");
                kaidaTheme_vector.push_back("Ka");

                kaidaTheme_vector.push_back("Dhi");
                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Gi");
                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("Ge");

                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("TiRa");
                kaidaTheme_vector.push_back("KiTa");
                kaidaTheme_vector.push_back("Dhi");
                kaidaTheme_vector.push_back("Ta");
                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("Ge");
                kaidaTheme_vector.push_back("Tra");
                kaidaTheme_vector.push_back("Ka");
                kaidaTheme_vector.push_back("Dhi");
                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Gi");
                kaidaTheme_vector.push_back("Na");

                kaidaTheme_vector.push_back("Ta");
                kaidaTheme_vector.push_back("Ke");
                kaidaTheme_vector.push_back("TiRa");
                kaidaTheme_vector.push_back("KiTa");
                kaidaTheme_vector.push_back("Ta");
                kaidaTheme_vector.push_back("Ge");
                kaidaTheme_vector.push_back("Tra");
                kaidaTheme_vector.push_back("Ka");
                kaidaTheme_vector.push_back("Ti");
                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Ki");
                kaidaTheme_vector.push_back("Na");
                kaidaTheme_vector.push_back("Dha");
                kaidaTheme_vector.push_back("Ge");

                kaidaTheme_vector.push_back("Na");
```

```
                                kaidaTheme_vector.push_back("Dha");
                                kaidaTheme_vector.push_back("TiRa");
                                kaidaTheme_vector.push_back("KiTa");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Ta");
                                kaidaTheme_vector.push_back("Dha");
                                kaidaTheme_vector.push_back("Ge");
                                kaidaTheme_vector.push_back("Tr");
                                kaidaTheme_vector.push_back("Ka");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("Gi");
                                kaidaTheme_vector.push_back("Na");

                                KaidaFastRoot = AppendThemeOfBols(kaidaTheme_vector,_tal,1);
                                break;
                                case 1:
                                kaidaTheme_vector.push_back("Dha");
                                kaidaTheme_vector.push_back("Ge");
                                kaidaTheme_vector.push_back("TiRa");
                                kaidaTheme_vector.push_back("KiTa");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("TiRa");
                                kaidaTheme_vector.push_back("KiTa");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("Gi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("Ti");
                                kaidaTheme_vector.push_back("Na");

                                kaidaTheme_vector.push_back("Ta");
                                kaidaTheme_vector.push_back("Ge");
                                kaidaTheme_vector.push_back("TiRa");
                                kaidaTheme_vector.push_back("KiTa");
                                kaidaTheme_vector.push_back("Ti");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("TiRa");
                                kaidaTheme_vector.push_back("KiTa");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("Gi");
                                kaidaTheme_vector.push_back("Na");
                                kaidaTheme_vector.push_back("Dhi");
                                kaidaTheme_vector.push_back("Na");

                                KaidaFastRoot = AppendThemeOfBols(kaidaTheme_vector,_tal,1);
                                break;

                        }
                        break;
        }
        kaidaTheme_vector.clear();
        return KaidaFastRoot;

}

int KaidaComposition::ChooseKaidaTheme(Tal _tal)
{
        srand((unsigned)time(0));


        int ThemeNumber;
        switch(_tal)
        {
        case eTEENTAL:
                ThemeNumber = (rand()%eTEENTHEME);
                break;
        case eJHAPTAL:
                ThemeNumber = (rand()%eJHAPTHEME);
                break;
        case eRUPAKTAL:
                ThemeNumber = (rand()%eRUPAKTHEME);
                break;
        }

        return ThemeNumber;

}

void KaidaComposition::KaidaDriver(Tal _tal, int NoOfPhrases)
{
```

```
        Node* ThekaTheme;
        Node* KaidaTheme;
        Node* KaidaTheme2;
        int KaidaThemeNumber;
        Node* KaidaThemeTemp;
        int ThemesForTal;

        //1)do the theka-----------------------------------------------------
        ThekaTheme = CreateThekaTree(_tal,true);
        this->KaidaStart = ThekaTheme;
        //this->KaidaStart->CompositionName = "Theka\n";
        //-------------------------------------------------------------------

        //2)do the theme-----------------------------------------------------
        //the the theme number and then the theme itself (it is in bhari/khali form)
        KaidaThemeNumber = ChooseKaidaTheme(_tal);
        KaidaThemeTemp = GetKaidaTheme(_tal,KaidaThemeNumber);

        //Calculate the number of themes would be required to fill up the tal
        ThemesForTal = NumberOfThemesForTal(KaidaThemeTemp,_tal);

        //put theme into appropriate tal before appending - This decides the relationship between theka and Kaida
        KaidaTheme = ChangeLayakari(KaidaThemeTemp,_tal,ThemesForTal,false);

        //Make a copy of this tree (KaidaTheme and KaidaTheme2 are not in bhari/khali form)
        KaidaTheme2 = copyTree(KaidaThemeTemp);

        KaidaTheme->CompositionName = "Kaida_Theme\n";
        this->KaidaStart->children.push_back(KaidaTheme);
        //-------------------------------------------------------------------


        //3)do the theme fast
        FullSpeedKaida(KaidaTheme,_tal);

        //4)do permutations
        PerformPalta(KaidaThemeTemp, _tal, NoOfPhrases);

        //5)do tihai
        TihaiComposition* TC = new TihaiComposition();
        TC->GenerateTihai(KaidaTheme2,_tal,5,true);
        while(TC->TihaiRoot == NULL){
                TC->GenerateTihai(KaidaTheme2,_tal,5,true);
        }
        this->KaidaStart->children.push_back(TC->TihaiRoot);

}

int KaidaComposition::NumberOfThemesForTal(Node* themeRoot, Tal _tal)
{
        int NoOfThemesForTal;

        this->BolsOfTree.clear();
        CollectBols(themeRoot);

        //figure out the amount of themes for tal (number of bols/beats in tal)
        NoOfThemesForTal = (this->BolsOfTree.size() / _tal);
        return NoOfThemesForTal;
}

//doubles speed of Theme
void KaidaComposition::FullSpeedKaida(Node* themeRoot, Tal _tal)
{
        //can change layakari to 2, but must work out the bharan
        //i.e. whether we need to fill out the tal
        Node* fastKaida;
        int NoOfThemesForTal;

        NoOfThemesForTal = NumberOfThemesForTal(themeRoot,_tal);

        fastKaida = ChangeLayakari(themeRoot,_tal,NoOfThemesForTal*2,false);
        fastKaida->CompositionName = "Full_Speed_Kaida\n";
        this->KaidaStart->children.push_back(fastKaida);

}

void KaidaComposition::PerformPalta(Node* root, Tal _tal, int NoOfPhrases)
{
        string BhariPalta;
        string KhaliPalta;
        vector<Node*> SecondLevelVariation;
        vector<Node*> SecondLevelVariationHalf;
        Node* rootCopy;
```

```
        int vectorCount;
        int i;
        string BhariPalta2;
        string KhaliPalta2;
        int NoOfBeatsToFill;
        int phraseCount;
        bool withRest;

//1st Level Variation-------------------------------------------------------------------------
        //1st Variation (AAAB-AAAB)
        DoVariation(root,"1,2,3,4","1,1,1,2","3,3,1,4",_tal, 1);

//-------------------------------------------------------------------------------------------
        for(phraseCount=1;phraseCount<(NoOfPhrases/2);phraseCount++)
        {
                //Perform variations for number of phrases
                BhariPalta = PaltaString(2,2); //2 nodes as there is only A and B
                KhaliPalta = FindKhaliStructure(BhariPalta,4);
                DoVariation(root,"1,2,3,4",BhariPalta,KhaliPalta,_tal, 2);
        }
//-------------------------------------------------------------------

//2nd Level Variation-------------------------------------------------------------------------
        //break a sentence into multiple parts and use them
        //strategy 1 - without pauses
        for(phraseCount=(NoOfPhrases/2);phraseCount<NoOfPhrases;phraseCount++)
        {
                SecondLevelVariationHalf.clear();

                if(NoOfPhrases-phraseCount == 1){withRest = true;}
                else{withRest = false;}
                //use the 1st sentence usually

                SecondLevelVariation = GetSubSentences(root->children[0],root->children[2], withRest);
                //creat a copy of the root
                rootCopy = copyTree(root);
                //add half of the SEcondlevel nodes to this copy (middle)
                for(i=0;i<(SecondLevelVariation.size()/2);i++){
                        SecondLevelVariationHalf.push_back(SecondLevelVariation[i]);
                }
                transformObj->InsertTransformation(rootCopy,3,SecondLevelVariationHalf);

                //add other half of secondlevel nodes to this copy (end)
                SecondLevelVariationHalf.clear();
                for(i=(SecondLevelVariation.size()/2);i<SecondLevelVariation.size();i++){
                        SecondLevelVariationHalf.push_back(SecondLevelVariation[i]);
                }
                transformObj->InsertTransformation(rootCopy,7,SecondLevelVariationHalf);

                this->BolsOfTree.clear();
                CollectBols(root);
                NoOfBeatsToFill = this->BolsOfTree.size() - (root->children[1]->children.size());
                BhariPalta2 = "";
                KhaliPalta2 = "";
                BhariPalta2 = PaltaString2(rootCopy,NoOfBeatsToFill, 0, 3);
                KhaliPalta2 = FindKhaliStructure(BhariPalta2,8);

                BhariPalta2 +=",-,-,-,-";
                KhaliPalta2 +="-,-,-,-";

                DoVariation(rootCopy,"1,2,3,4,5,6,7,8",BhariPalta2,KhaliPalta2,_tal, 1);

        }
        //-------------------------------------------------------------------------------------

}


vector<Node*> KaidaComposition::GetSubSentences(Node* rootBhari, Node* rootKhali, bool withRests)
{
        vector<Node*> SubSentenceRoots;
        int maxBound;
        int rNo1;
        int rNo2;
        int upperBound;
        int lowerBound;
        Node* rootCBhari;
        Node* rootDBhari;
        Node* rootCKhali;
        Node* rootDKhali;
        Node* tempNode1;
        Node* tempNode2;
```

```
        int i;

        srand((unsigned)time(0));
        //define max as the no of children of root
        maxBound = (rootBhari->children.size())-1;

        rNo1 = 0+int(maxBound*rand()/(RAND_MAX + 1.0));
        rNo2 = 0+int(maxBound*rand()/(RAND_MAX + 1.0));

        //if they are the same, then do again;
        while(rNo1==rNo2){
                rNo2 = 0+int(maxBound*rand()/(RAND_MAX + 1.0));
        }

        //these are the range of children
        lowerBound = min(rNo1,rNo2);
        upperBound = max(rNo1,rNo2);

        //the roots of our sub sentences
        rootCBhari = new Node(1,"C");
        rootDBhari = new Node(1,"D");

        rootCKhali = new Node(1,"C'");
        rootDKhali = new Node(1,"D'");

        //append the appropriate bols to the relevant trees
        for(i=0;i<=maxBound;i++)
        {
                if(i>=lowerBound && i<= upperBound){
                        tempNode1 = copyTree(rootBhari->children[i]);
                        tempNode2 = copyTree(rootKhali->children[i]);

                        rootCBhari->children.push_back(tempNode1);
                        tempNode1->parent = rootCBhari;
                        rootCKhali->children.push_back(tempNode2);
                        tempNode2->parent = rootCKhali;

                }
                else{
                        tempNode1 = copyTree(rootBhari->children[i]);
                        tempNode2 = copyTree(rootKhali->children[i]);

                        rootDBhari->children.push_back(tempNode1);
                        tempNode1->parent = rootDBhari;
                        rootDKhali->children.push_back(tempNode2);
                        tempNode2->parent = rootDKhali;
                }
        }


        //the D tree (Bhari and Khali) should have a pause
        if(withRests){
                //choose a bol from the tree
                rNo1 = 0+int((rootCBhari->children.size()-1)*rand()/(RAND_MAX + 1.0));
                while(rootCBhari->children.empty() == true && rootCBhari->children[rNo1]->children.empty() == true){
                        rNo1 = 0+int((rootCBhari->children.size()-1)*rand()/(RAND_MAX + 1.0));
                }
                rootCBhari->children[rNo1]->children[0]->value = "S";
                rootCKhali->children[rNo1]->children[0]->value = "S";
        }

        SubSentenceRoots.push_back(rootCBhari);
        SubSentenceRoots.push_back(rootDBhari);
        SubSentenceRoots.push_back(rootCKhali);
        SubSentenceRoots.push_back(rootDKhali);

        return  SubSentenceRoots;

}

void KaidaComposition::DoVariation(Node* root,string structuralIndex, string structuralChangeBhari,string structuralChangeKhali,Tal _tal, int
{
        Node* newRootBhari;
        Node* newRootKhali;
        int NoOfThemesForTal;
        string BhariName = "Variation ";
        string KhaliName = "Variation ";

        vector<string> WholeVariation;
        int i;

        newRootBhari = copyTree(root);
        newRootKhali = copyTree(root);
```

```
        //The first iteration has the AAAB-AAAB variation
        //i.e. A1,A1,A1,B1 - A2,A2,A1,B2

    transformObj->transformation(structuralIndex,structuralChangeBhari,newRootBhari,false);
        transformObj->transformation(structuralIndex,structuralChangeKhali,newRootKhali,false);

        //Must figure out the bharan i.e. whether we need to fill the tal
        //The ChangeLayakari function will put the bols in the appropriate phrase structure
        NoOfThemesForTal = NumberOfThemesForTal(newRootBhari,_tal);

        this->BolsOfTree.clear();
        CollectBols(newRootBhari);

        for(i=0;i<this->BolsOfTree.size();i++)
        {
                WholeVariation.push_back(this->BolsOfTree[i]);
        }
        this->BolsOfTree.clear();
        CollectBols(newRootKhali);
        for(i=0;i<this->BolsOfTree.size();i++)
        {
                WholeVariation.push_back(this->BolsOfTree[i]);
        }
        newRootBhari = ChangeLayakari(WholeVariation,_tal,NoOfThemesForTal*2,true);
        newRootBhari->CompositionName = "_(Variation)\n";
        this->KaidaStart->children.push_back(newRootBhari);
}

string KaidaComposition::PaltaString2(Node* root,int NoOfBeats, int intFrom, int intTo)
{
        //there should be 4 children but in case there isn't
        vector<int> NoOfBols;
        std::stringstream structuralChange;
        int i;
        int beatCount = 0;
        bool gotSequence = true;
        int randomNumber;
        string temp;
        string strCh="";
        string strTemp;

        for(i=intFrom;i<=intTo;i++){
                NoOfBols.push_back(root->children[i]->children.size());
        }

        while(gotSequence){

                randomNumber = 0+int((NoOfBols.size())*rand()/(RAND_MAX + 1.0));
                beatCount = beatCount + NoOfBols[randomNumber];

                strTemp = itos2(randomNumber+1);
                strCh += strTemp;
                strCh += ",";
                strTemp = "";

                if(beatCount == NoOfBeats){
                        gotSequence = false;
                        randomNumber = 2;
                        strTemp = itos2(randomNumber);
                        strCh += strTemp;
                }
                else if(beatCount > NoOfBeats){
                        beatCount = 0;
                        strCh = "";
                }
                //else we are lower than the NoOfBeats, so loop
        }
        return strCh;

}
string KaidaComposition::PaltaString(int phraseNo, int noOfNodes)
{
        int i;
        string palta = "A,";
        int randomNumber;
        string childA;
        string childB;
        string childC;
        string childD;
        string childE;
        vector <string> paltaVector;
        string structuralChange="";
        int max;
```

```
        max = ((noOfNodes+1)*(noOfNodes+1));

        for(i=0;i<phraseNo;i++){
                randomNumber = 0+int(max*rand()/(RAND_MAX + 1.0));
                if(randomNumber >= 0 && randomNumber <= 3)
                {
                        palta += "A,";
                }
                else if(randomNumber >= 4 && randomNumber <= 7)
                {
                        palta += "B,";
                }
                else{i=i-1;}

        }
        palta += "B";
        paltaVector = Split(",",palta);

        childA = "1";
        childB = "2";
        childC = "3";
        childD = "4";
        childE = "5";

        srand((unsigned)time(0));
        for(i=0;i<paltaVector.size();i++)
        {
                if(paltaVector[i] == "A"){
                        structuralChange += childA;
                        structuralChange += ",";
                        randomNumber = 1+int(noOfNodes*rand()/(RAND_MAX + 1.0));
                        if(randomNumber == 1){childA = "1";}
                        else{childA = "3";}
                }
                else if(paltaVector[i] == "B"){
                        structuralChange += childB;
                        structuralChange += ",";
                        randomNumber = 1+int(noOfNodes*rand()/(RAND_MAX + 1.0));
                        if(randomNumber == 1){childB = "2";}
                        else{childB = "4";}
                }
        }
        return structuralChange;

}

string KaidaComposition::FindKhaliStructure(string structuralChange, int max)
{
        vector<string> sChange;
        int i;
        sChange = Split(",",structuralChange);

        int nodeValue;
        std::ostringstream newStructuralChange;
        int newValue;

        for(i=0;i<sChange.size();i++)
        {
                nodeValue = atoi(sChange[i].c_str());

                if(nodeValue > (max/2)){
                        newValue = nodeValue - (max/2);
                }
                else if(nodeValue <= (max/2)){
                        newValue = nodeValue + (max/2);
                }

                newStructuralChange << newValue;
                newStructuralChange << ",";
        }

        return newStructuralChange.str();
}
```

## TihaiComposition.cpp

```
/***************TihaiComposition.cpp************************************************
This class uses the tree that is passed to it and enumerates the bols giovene the
criteria of the tihai to generate a new tree (using the Tal phrase structure)
of three Pallas.
*******************************************************************************/
```

```
#include "TihaiComposition.h"

TihaiComposition::TihaiComposition()
{
	TihaiRoot = NULL;
}
TihaiComposition::~TihaiComposition(){}

//This method will use the tree passed to generate the tihai
void TihaiComposition::GenerateTihai(Node* root, Tal _tal, int layakari, bool withBharan)
{
	int NoOfSentences;
	int NoOfBeats;
	int layakari1;
	int NoOfBeatsToFill;
	int UnitsInPalla;
	int QuadraticSentence;
	int beatCount=0;
	int randomNumber;
	bool gotSequence=true;
	string strCh="";
	string strTemp;
	vector<int> NoOfBols;
	vector<Node*> VectorBols;
	vector<string> TihaiBols;
	vector<Node*> TihaiParentBols;
	Node* BharanTree;
	Node* TihaiTree;
	int i;
	int j;
	int k;
	int l;
	int UnitsOfPause = 0;
	int tempUnitsInPause;
	bool withPauses = false;
	bool loopControl = true;

	//work out number of units in palla

	//layakari = NoOfBols / Tal
	//ALSO NoOfBols = number of beats to fill

	this->BolsOfTree.clear();
	CollectBols(root);
	layakari1 = ((this->BolsOfTree.size()*2) / _tal);

	//Calculate original number of beats
	if(withBharan || _tal%2 != 0){
		if(_tal%2 != 0){
			NoOfBeatsToFill = _tal;
		}
		else{
			NoOfBeatsToFill = _tal/2;
		}
	}
	else{NoOfBeatsToFill = _tal;}

	//Calculate wheather a Bedum or Dumbar Tihai is generated
	while(((layakari1*NoOfBeatsToFill)+1)%3 != 0){
		if(NoOfBeatsToFill > 100)
		{
			//must use pauses.
			NoOfBeatsToFill = _tal;
			withPauses = true;
			break;
		}
		NoOfBeatsToFill = NoOfBeatsToFill*2;
	}

	//Actually generate the number of units required
	UnitsInPalla = CalculateUnitsInPalla(withPauses,layakari1,NoOfBeatsToFill);

	//To choose the bols, use the method as in KaidaComposition::GetSubsentences
	//of gathering the children of root and the number of bols there are and selecting
	//one at random in a loop until the UnitsInPalla is met

	for(NoOfSentences=0;NoOfSentences<root->children.size();NoOfSentences++){
		for(NoOfBeats=0;NoOfBeats<root->children[NoOfSentences]->children.size();NoOfBeats++){
			NoOfBols.push_back(root->children[NoOfSentences]->children[NoOfBeats]->children.size());
			VectorBols.push_back(root->children[NoOfSentences]->children[NoOfBeats]);
		}
	}
```

```
srand((unsigned)time(0));
int loopCount =0;

//choosing parts of the tree (transformation - randomly chosen)
while(gotSequence){
        randomNumber = 0+int((NoOfBols.size())*rand()/(RAND_MAX + 1.0));

        beatCount = beatCount + NoOfBols[randomNumber];

        TihaiParentBols.push_back(VectorBols[randomNumber]);

        if(beatCount == ((UnitsInPalla)-1)){
                gotSequence = false;
        }
        else if(beatCount > (UnitsInPalla)-1){
                beatCount = 0;
                TihaiParentBols.clear();
        }

        loopCount = loopCount + 1;
        //we are probably in an infinate loop for the random number gen
        if(loopCount >=100){
                beatCount = 0;
                TihaiParentBols.clear();
                layakari1 = layakari1+1;
                UnitsInPalla = CalculateUnitsInPalla(withPauses,layakari1,NoOfBeatsToFill);
                loopCount = 0;
                //return;
        }
        //else we are lower than the NoOfBeats, so loop
}

//iterate through a tree and append firstly the bols of the root and then the VectorBol
for(k=0; k<3;k++)
{
        for(i=0;i<TihaiParentBols.size();i++)
        {
                for(j=0;j<TihaiParentBols[i]->children.size();j++)
                {
                        //this->BolsOfTree.push_back(TihaiParentBols[i]->children[j]->value);
                        TihaiBols.push_back(TihaiParentBols[i]->children[j]->value);
                }
        }
        //take off the UnitsOfPause

        TihaiBols.push_back("Dha");
        if(withPauses){
                //add on the blanks
                for(tempUnitsInPause=0;tempUnitsInPause<UnitsOfPause;tempUnitsInPause++){
                        TihaiBols.push_back("S");
                }
        }
}

//now must decide how many trees I need to include the bharan and tihai
//3 ways:
//b) no bharan, 1 tal tihai
//if NoOfBeatsToFill = _tal

//take off the last Bol as this will be the resolving sam
TihaiBols.pop_back();

if(withBharan){
        //a) 1 tal - half bharan, half tihai
        //if NoOfBeatsToFill = _tal/2

        if(NoOfBeatsToFill == _tal/2){
                BharanTree = ChangeLayakari(this->BolsOfTree,_tal,layakari1,true);
        }
        else{
                BharanTree = ChangeLayakari(this->BolsOfTree,_tal,layakari1,false);
        }

        TihaiTree = ChangeLayakari(TihaiBols,_tal,layakari1,true);
        TihaiTree->CompositionName = "␣Tihai\n";
        BharanTree->CompositionName = "␣Bharan\n";
        BharanTree->children.push_back(TihaiTree);
        this->TihaiRoot = BharanTree;
}
else{
                //if a larger tree is required (if _tal*layakari > (TihaiBols.size-1))
                TihaiTree = AppendVectorOfBols(TihaiBols,_tal,layakari1);
                TihaiTree->CompositionName = "␣Tihai\n";
```

```
                        this->TihaiRoot = TihaiTree;
        }

        //must now add the Sam
        Node* SamNode = new Node(2,"Sam");
        BolNode* SamBol = new BolNode(3,"Dha");
        SamNode->children.push_back(SamBol);
        this->TihaiRoot->children.push_back(SamNode);
}

int TihaiComposition::CalculateUnitsInPalla(bool _withPauses, int _layakari, int _NoOfBeatsToFill)
{
        int UnitsOfPause;
        int UnitsInPalla;
        bool loopControl = true;

                if(_withPauses){
                UnitsOfPause = 1;
                while(loopControl)
                {
                        //3P + 2U = (Layakari*NOOfBeatsToFill+1)/3
                        //Find an appropriate Units of pause that will work
                        if((((_layakari*_NoOfBeatsToFill)+1) - (2*UnitsOfPause)) % 3 != 0){
                                UnitsOfPause++;
                        }
                        else{
                                UnitsInPalla = ((((_layakari*_NoOfBeatsToFill)+1) - (2*UnitsOfPause)) / 3);
                                loopControl = false;
                        }
                }

        }
        else{
                UnitsInPalla = ((_layakari*_NoOfBeatsToFill)+1)/3;
        }

        return UnitsInPalla;
}
```

# Bibliography


[1] Anne Abeille and Owen Rambow. *Tree Adjoining Grammars*, chapter Tree Adjoining Grammars: An Overview, pages 1–69. CSLI, 2000. Leland Stanford Junior University.

[2] Dominic Alldis. *Jazz Piano Improvisation*, chapter Major Scale Threory Applied, page 74. Hal Leonard Corporation, 2003.

[3] Benard Bel. Pattern grammars in formal representations of musical structures. In *Workshop on AI and Music*, pages 113–42. 11th International Joint Conference on Artificial Intelligence, June 1989.

[4] Benard Bel. Welcome to the bol processor 2, 1999. `http://www.lpl.univ-aix.fr/~belbernard/music/`.

[5] David Bernsterin and Tom Strychacz. Beyond mainstream: An interdisciplinary study of music and the written word. *Language and Learning Across the Disciplines*, 1(3):49–66, August 1996.

[6] Richard Boulanger. *The CSound Book*. Massachusettes Institute of Technology, 2000.

[7] Noam Chomsky. *Syntactic Structures*. Mouton and Co., The Netherlands, 6 edition, 1966.

[8] David Courtney. The cadenza in north indian tabla. *Percussive Notes*, 32(4):54–64, August 1994.

[9] David Courtney. *Advanced Theory of Tabla*. Sur Sangeet Services, 2000.

[10] David Courtney. *Learning the Tabla*. Mel Bay Publications, 2001.

[11] David Courtney. *A Focus on the Kaidas of Tabla*. Sur Sangeet Services, 2002.

[12] Alfonso Ortega de la Puente, Rafael Sanchez Alfonso, and Manuel Alfonseca Moreno. Automatic composition of music by means of grammatical evolution. In *APL '02: Proceedings of the 2002 conference on APL*, pages 148–155, New York, NY, USA, 2002. ACM Press.

[13] C. DORAN, B. HOCKEY, A. SARKAR, B. SRINIVAS, and F. XIA. Evolution of the xtag system, 2000.

[14] John T Grinder and Suzette Haden Elgin. *Guide To Transformational Grammar: History, Theory, Practice*. Holt, Rinehart and Winston Inc., 1973.

[15] Swar Systems Inc. First steps with taalwizard: Swarshala v2 introductory notes.

[16] A. Joshi and Y. Schabes. Tree-adjoining grammars. *Handbook of Formal Languages*, 3:69–124, 1997. Springer, Berlin.

[17] Ustad Sharafat Khan. *Enter the World of Tabla*. Indian Classical Music Centre, 205, Balestier Road, Ruby Plaza B1-11 Singapore 329682, 3 edition, 2000.

[18] James Kippen. *The Tabla of Lucknow*. Mahohar Publishers and Distributors, 2005.

[19] Jim Kippen. Computational techniques in musical analysis. *BICA*, 4, March 1986.

[20] Jim Kippen and Benard Bel. The identification and modelling of a percussion 'language', and the emergence of musical concepts in a machine-learning experimental set-up. *Computers And the Humanities*, pages 199–214, 1989.

[21] Jim Kippen and Benard Bel. Modelling music with grammrars: Formal language representation in the bol processor. *Computer Representations and Models in Music*, pages 207–38, 1992. A. Marsden and A. Pople., Eds., Academic Press, London, 1989a.

[22] Howard Lasnik. *Syntactic Structure Revisited*. Massachusetts Institute of Technology, 2000.

[23] Jerry Leake. *Volume II: Indian Influence (Tabla Perspectives)*, chapter Appendix C: Many Roles of the Tabla Player, page 1. Rhombus Publishing, 3rd edition, 1996.

[24] Candito Marie-H. Generating an ltag out of a principle-based hierarchical representation. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 342–344, Morristown, NJ, USA, 1996. Association for Computational Linguistics.

[25] Jon McCormack. Grammar based music composition, 1996.

[26] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, page 103, 1997.

[27] Thomas Scovel. *Psycholinguistics*, chapter Acquisition. Oxford University Press., 1998.

[28] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[29] Gareth Williams. *Linear Algebra with Applications*. Jones and Bartlett Publishers, 2001.